REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 24/May/2001	2. REPORT TYPE THESIS	3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE AN OBJECT DESCRIPTION LANGUAGE FOR DISTIBUTED DISCRETE EVENT SIMULATIONS	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) MAJ ANDREWS HAROLD G	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) TUFTS UNIVERSITY	8. PERFORMING ORGANIZATION REPORT NUMBER CI01-75
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) THE DEPARTMENT OF THE AIR FORCE AFIT/CIA, BLDG 125 2950 P STREET WPAFB OH 45433	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

13. SUPPLEMENTARY NOTES

14. ABSTRACT

20010720 039

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE		435	19b. TELEPHONE NUMBER *(Include area code)*

An Object Description Language for Distributed Discrete Event Simulations

by

Harold Gregory Andrews II, Major, USAF

Submitted to the Department of Electrical Engineering and Computer Science on 13 April 2001, in partial fulfillment of the requirements for the Degree of Doctor of Philosophy of Computer Science

Abstract

Digital simulation is a useful tool for developing a better understanding of physical or hypothetical systems. It has been used with great success since the advent of the digital computer in such varied fields as weather prediction, planning military operations, and training. As digital computers become more capable and network communications systems more prevalent, the notion of synergistically combining the two to perform distributed simulation has led to some tremendous improvements in simulation speed and fidelity.

This dissertation describes a new programming language that is useful in creating distributed discrete event simulations without burdening simulation developers with the difficult and error-prone task of synchronizing nodes in a distributed simulation. Developers can instead focus on specifying the behavior of the objects in the virtual environment with little effort devoted to lower level concerns.

The language structure follows the notions of stimulus-response and completely isolates simulation object instances from each other. Inter-object communication occurs solely through message passing. Several example applications are described.

Thesis Advisor: Professor David W. Krumme, Associate Professor of Computer Science

Technical Supervisor: Dr. Owen L. Deutsch, Senior Member of the Technical Staff

An Object Description Language for Distributed Discrete Event Simulations

A dissertation submitted by

Harold Gregory Andrews II, Major, USAF

MS Mathematics, University of Texas, San Antonio, 1997
MBA Management, Rensselaer Polytechnic Institute, 1992
BS Mathematics, Northeastern University, 1988
BS Computer Science, Northeastern University, 1988

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment for the requirements for the degree of

Doctorate of Philosophy in Computer Science

TUFTS UNIVERSITY

May 2001

Thesis Advisor: Associate Professor David W. Krumme
Draper Technical Supervisor: Dr. Owen L. Deutsch
Thesis Committee Member: Associate Professor Anselm Blumer
Thesis Committee Member: Associate Professor Robert J. K. Jacob

ACKNOWLEDGEMENT

13 April 2001

I first and foremost acknowledge the intervention from the Almighty God who, through His divine providence, saw fit to allow me to attend school and work on this project.

Next, I thank my wife and best friend Barbara whose love, devotion, and willingness to deal with much in terms of caring for our children, what seems to have been more sickness than health, and my vacant stares while I pondered some troubling technical triviality, freed me from having to fret over managing many of our household affairs; they have indeed been in good hands.

Special thanks go also to my Tufts University thesis advisor, Professor David Krumme, and my Draper Laboratory technical supervisor, Dr. Owen Deutsch for their help, guidance, and insight as I worked on this dissertation and the related project. In addition, thanks to my thesis committee members Professors Anselm Blumer and Robert Jacob for their willingness to wade through the musings contained herein, and quiz me about them. I would also like to thank Professor Karen Panetta for her participation as an examiner during my orals, and for the observations, comments and insights she offered; they were very helpful.

Additional thanks go to current and former personnel at the Air Force Institute of Technology, office of Civilian Institution Programs: Colonel Paul Copp, for fighting for me to go to school, when the Air Staff was opposed to the idea; Majors Ralph Tolle and David Schluckebier for acting as my voice at AFIT; Captain Rick Sutter, who suggested a novel way of dealing with administrative difficulties. Also, thanks go to the faculty of the computer science department at the United States Air Force Academy who were willing to take a chance on an unknown.

In no particular order I also acknowledge, and humbly thank the following for their prayers, advice, support, and words of encouragement: my mother, Jill Andrews, for her help beyond the call of duty during the aforementioned periods of sickness; my father, Harold Andrews, for providing me with an environment in which I was free to explore my various interests; my mother-in-law, Veronica Vogt, for her help, prayers, encouragement and mastery of the English language; my father-in-law, Kenneth Vogt who helped me to keep my sense of humor; my former high school math teacher, Eugene Kaczowka, who in many ways started this ball rolling lo those many years ago; Major William Szarek, whose dedication to the notion of service above self provided me with as a model to emulate; Majors Michael Ward and Barrie Wheeler, for helping me to see how things really are; Lieutenant Colonel Skip Youngberg for giving me a well deserved and much needed kick in the pants; Colonel Larry Carr and Lieutenant Colonel Richard Saint Pierre, who provided me with counsel, wisdom, and the perspective of their experience; Robert Balusek, Dr. John Romo, and Dr. Stephen Welstead, for their advice in matters of higher education; and to those unmentioned by name here who through their prayers have allowed this work to come to a merciful conclusion.

This thesis was prepared at the Charles Stark Draper Laboratory, Inc., under projects 13025 (Multi-Agent Collaboration Services), 15052 (SOS Large Scale Analysis Capability), 18527 (C4I Laboratory Capability Development), and 13360 (Tactical Node).

Harold Gregory Andrews II, Major, USAF

Contents

Tables

Figures

"In theory, there is no difference between theory and practice; in practice, there is."

– Chuck Reid

Chapter 1. Rationale and background

1.1. Overview

Simulation in recent years has become increasingly important in understanding natural systems. In particular, digital simulation provides a method of hypothesis testing, training, and planning that might otherwise be prohibitive because of either cost or risk. For instance, it's cheaper and safer to train nuclear power plant operators on digital simulations than on fully functional equipment. The same claim can be made for manned space flight operations, as well as, to a somewhat lesser extent, aircrew training.

Industry has become increasingly interested in simulation technologies as the cost of developing prototypes has dramatically increased. An example of this would be the development of the Boeing 777 airliner. Boeing designed and digitally tested the 777 for performance and manufacturability before ever building the first prototype. These tests were conducted using digital simulation (Boeing 2001).

The military has also increasingly relied upon simulation to provide insight into deployment, operations and support plans, and was instrumental in developing early flight simulators to provide additional training hours to pilots. More recently, ground and naval combatants have used digital simulation for developing tactics and training purposes. Simulation systems to assess the overall effectiveness of new weapons and tactics are also in widespread use throughout the U.S. Department of Defense (DoD). It has become so important that in 1991, DoD established the Defense Modeling and Simulation Office (DMSO) to coordinate simulation activities throughout the department (DMSO 2001).

It was in the arena of military simulation that the notion of this dissertation took shape. In 1997, the author was involved in a simulation exercise to ascertain the operational effectiveness of non-lethal weapons under a variety of scenarios. The client organization insisted these exercises use a simulation system known as the Joint Tactical Simulation (JTS)[1]. JTS did not have the capability to simulate the effects these new weapon systems were thought to produce in the targets. Specifically, a target was either alive or dead. Since the purpose of the weapon system was to coerce a behavior in the target, rather than to outright kill it, there was a substantial effort involved in trying to get the simulation results to reflect the hypothesized effects.

If there had been the capability to change the behavior of various objects in the simulation that end users could modify while maintaining the overall system integrity, then the effects of these new weapons could have been introduced quite simply. It would not have required re-verifying the entire simulation system, but only the altered portions. This became the ultimate motivation for providing a framework for describing simulation object behavior in an extensible manner.

The result of this line of thought, and the object of this dissertation, is the Simulation Object Description Language (SODL – pronounced any way the reader prefers). SODL is an object oriented language, in that it has the standard ability to inherit behavior from parent classes. It is also completely event driven, in that object instances can only communicate with each other via message passing. This provides for the possibility of allowing the simulation to be transparently distributed across a heterogeneous network of computers while freeing the developer from actually producing the code to perform run-time synchronization, networking, message sequencing and delivery required to ensure that distributed simulations process messages in the proper order. Having stated this, we should point out that the actual run-time system developed to test and support SODL programs runs only on a single processor at the time of this writing. We took great care to ensure that it could easily be modified to a fully distributed implementation.

Digital simulation is performed in any of a number of fashions, some of which will be discussed herein at some length. A digital simulation is broken down into a finite sequence of events. When each event is handled, the state of the simulation changes in some fundamental way. The difficult part for some types of simulation is ensuring that these events are handled in the proper order. This is not particularly difficult for

[1] JTS was combined with the Joint Conflict Model (JCM) to form the Joint Conflict and Tactical Simulator (JCATS). These simulators were all developed at the Lawrence Livermore National Laboratory, Livermore

simulation systems that run in only one process. In fact, events normally have some sort of time stamp associated with them, which, if these events are generated out of their intended processing order, can be placed into a priority queue or some other sorting mechanism to ensure that they are processed in the correct order.

The situation is not nearly as clear-cut in distributed simulation. On the one hand, if multiple processors can be brought to bear upon a simulation problem, then an answer can be arrived at in a shorter period of time than would otherwise be possible. On the other hand, issues such as network latency, lost packets, or packets received out of order, and clock drift between CPUs in a distributed system can cause desynchronization between those nodes to occur, allowing events to be processed out of order. When events are processed out of order, it creates what is known as a causality error (Fujimoto 1990).

There are two basic approaches for dealing with these causality errors in distributed simulation. *Conservative synchronization* (Bryant 1977), (Chandy 1979, 1981) seeks to avoid creating these errors by blocking processing on nodes in a distributed simulation system until additional processing can proceed without the risk of creating a causality error. Alternatively, *optimistic synchronization* (Jefferson 1982, 1985a, 1987) provides a mechanism for detecting and recovering from these causality errors. It does this through state saving, allowing for the possibility of recovery from potential causality errors. When one is actually encountered, a cascading rollback and event revocation algorithm is used to restore the simulation system to a state whereby processing the event is temporally consistent with the state of the simulation. Since it does not block, it can provide a significant speed-up in simulation execution (Fujimoto 1990). The SODL implementation discussed herein uses an optimistic simulation engine to perform process synchronization.

Neither of these approaches is appropriate for all simulation systems. Systems where there is a high degree of interdependence between objects within the simulation may lend themselves to a conservative approach. Optimistic synchronization may be appropriate for systems in which the components are interdependent to a limited extent. (Fujimoto 1990) provides a more extensive analysis of the benefits and limitations of each approach.

CA. For more information refer to (LLNL 1998).

The primary focus of this dissertation is the structure of the SODL language. Even though we implemented a sequential (i.e. non-distributed) runtime system, its usefulness stems mainly from its ability to test the SODL language structure. The primary rationale for this approach is that the body of work on this subject continues to grow. The run-time system provided is intended to act as a framework for incorporating these new results later, and should not be considered a final product.

1.2. Alternative approaches

There are a number of different approaches available to simulation system developers. What follows is a list of the various types of approaches that are currently in use, their strengths, weaknesses, and some examples. Table 1-1 provides a brief overview of these alternative approaches.

Approach	*Description*
Full-Implementation	All aspects of the simulation engine are built from scratch for a particular system.
Modular System	Portions of the simulation engine are linked into a final executable to perform some of the more mundane aspects of simulation such as message delivery and node synchronization. Calls to these library functions are still needed to make use of the simulation engine.
Simulation Languages	The simulation engine is provided as a run-time environment in which simulation developers describe the behavior of the simulation objects. It is rarely, if ever, necessary for a developer to make any direct calls to the simulation engine for messaging, sequencing, or synchronization.

Table 1-1 Alternative Simulation Approaches

1.2.1. Full implementations

Early digital simulations were implemented first in low-level machine languages, and later, as compilers became available, in traditional structured programming languages such as Fortran, C, and others. All aspects of the simulation engine, from event sequencing through node synchronization were written and custom tailored to the specific system implementation. The JTS and JCM simulators mentioned earlier evolved from simulation systems developed in the 1970's (LLNL 1998). One can open any journal on simulation to find this approach in use, even to this day.

More recently, beginning in the late 1980's, object oriented programming languages such as C++, Modula-2, and Smalltalk made certain aspects of simulation development somewhat easier and significantly more intuitive. This increased ease did not extend to the core simulation engine, however. Still remaining were the complexities associated with process synchronization in distributed simulation systems.

This approach has the advantage that there is a substantial amount of control developers have over optimizing system performance for a particular task. This flexibility, however, comes at a rather substantial cost in coding and debugging effort. This is particularly the case for distributed simulation, where node synchronization issues require extensive and complex code to properly address.

Some examples of full simulation system implementations are listed in Table 1-2.

Name & Producer	***Description***
Joint Conflict and Tactical Simulator Lawrence Livermore National Laboratory (LLNL 1998)	A sequential simulation system with a distributed user interface, allows multiple operators to perform combat exercises to test new tactics and weapon systems. Used also for training purposes. Written primarily in C++. Users include the U.S. Departments Defense, Energy, and Treasury, as well as some international users.
Joint Simulation Systems (JSIMS) Defense Modeling & Simulation Office JSIMS Program Office	A distributed simulation system that is currently under development, using the High Level Architecture (HLA) Run Time Infrastructure (RTI) as a packet transport mechanism. It is intended to provide an extensive simulation capability for conducting training and analysis and doctrine development.
Diffract MM Research Tucson, AZ	Diffract is an optical simulation system used for simulating coherent light through optical systems. It can be used to simulate optical aberrations and interference patterns in optical equipment (Mansuripur 1997).

Table 1-2 Some simulation systems used for analysis or training purposes

In addition to the specific simulation systems mentioned in Table 1-2, there are numerous other simulation packages designed for fields as diverse as computational fluid dynamics to manufacturing. Others are designed to provide insight into the motions of stars and galaxies, to the workings of the smallest known particles. Many of these custom simulation systems, though very capable in what they do, are intended for a specific use that does not readily extend to other uses.

Another area in which simulation is gaining some popularity is in the field of interactive digital entertainment (by which we mean computer video games). Most of these are built on proprietary special purpose simulation engines that facilitate event sequencing. Many of the newer games provide the ability to perform distributed game play over the Internet or via direct dial-up. Some recent examples of these interactive games are listed in Table 1-3. Distributed game play has become increasingly popular, as more households are equipped with dial up and broadband Internet access. There are even massively parallel games in which an entire persistent virtual world has been created. Players can come and go as they please, joining forces with other players to battle against computer-controlled entities, or against other players. With faster Internet access becoming more widely available, online gaming is likely to continue growing in popularity and the games themselves will also continue to increase in their complexity as well.

Name & Publisher	***Description***
The Sims Electronic Arts	A successor to the popular SimCity. The Sims allows players to control people in the virtual environment as they interact with others. Virtual characters are affected by the inputs their real life controllers
Battlezone and Battlezone II Activision	Players control a virtual army in an immersive 3-D environment and can issue orders to their subordinate units, manage resources and engage in combat all from a first person perspective. Multi-player modes are available to play over the Internet and over a LAN.
Flight Simulator 2000 Pro Microsoft	Players can take control of one of a variety of aircraft, each modeled to resemble the actual performance and handling characteristics of the real world counterpart. Microsoft also provides the capability to download from the Internet current real-world weather conditions.
Ultima Online Electronic Arts	A massively parallel virtual and persistent world where players control a virtual character who can fight wars with other player, build structures, or even complete cities. Any changes the players make persist so that others may interact with those changes.

Table 1-3 Current popular games using simulation technologies

1.2.2. Modular simulation systems

It is sometimes possible to encapsulate portions of the central simulation engine functions in a collection of libraries, such as event sequencing and some rudimentary message delivery. Some libraries may perform network communications and synchronization required in distributed simulation. Others provide additional functionality defining, for instance, the behavior of simulation objects incorporated into a separate development initiative. The intent of these approaches is to use the libraries with a more commonly available programming language such as C/C++, Fortran, or Java. Other approaches, such as CORBA, require programmers to write portions of the simulation system in a different language, the Interface Description Language (IDL) in the case of CORBA. A compiler translates this code into a more common object oriented language. Standard compilers then compiles and links with the run-time infrastructure in the library.

Name & Developer	***Description***
Modular Semi-Autonomous Forces (ModSAF) United States Army Simulation, Training, and Instrumentation Command (STRICOM)	A semi-distributed simulation system which has a number of military entities described in detail as a series of C modules. Simulation entity behavior is intended to mimic real world behavior to facilitate training and analysis for military operations. Simulation state is stored in a central database that is accessed via run-time engine (STRICOM 1999).
High Level Architecture (HLA) Defense Modeling and Simulation Organization (DMSO)	HLA provides a run-time infrastructure to facilitate distributed simulation. It primarily deals with network communications, not synchronization, which must be performed in custom simulation engines.
Common Object Request Broker: Architecture (CORBA) The Object Management Group	CORBA is a distribute object system that can be applied to simulation. As in HLA, a synchronization mechanism must be provided to ensure events are processed chronologically.

Table 1-4 Examples of modular simulation systems

While this approach does free developers from writing the complex code associated with these functions, there is still a great deal of interfacing with those libraries that must take place. Considerable time must be spent on the part of the developer to actually learn the API for the library. It is also usually necessary, in order to get the proper results out of the system, for the developer to have a keen understanding of the manner in which the routines or classes function (particularly in the case of libraries which provide object behavior descriptions and those that provide synchronization in distributed simulation).

Some examples of systems that provide this modular approach are listed in Table 1-4.

1.2.3. 4th generation simulation programming languages

Recent years have seen the development of a number of special purpose simulation languages. Each of these has been designed to fill a certain niche. The majority of these languages have been sequential systems, by which we mean that they are intended to operate on only one host computer (i.e. they are not distributed). They can generally be broken down into two general categories: continuous time simulators (CTS) and discrete event simulators (DES). The biggest difference between CTS and DES systems regard how events are generated. CTS events are generated sequentially, while DES systems may generate their events out of order. Most simulation languages are sequential systems, not capable of operating in a distributed manner. Some of these sequential systems are listed in Table 1-5.

Name & Developer	***Description***
Simulink The MathWorks	Provides a graphical user interface to perform continuous time simulation of systems governed differential equations. Fully interoperable with Matlab, and extensible with custom routines from other programming languages. (MathWorks 2001)
ProModel ProModel Corp.	Graphical tool for simulating and analyzing processes. Extensive analysis tools are provided as well as the ability to export data to third party spread sheets for custom analysis.
SOAR Carnegie Mellon University, University of Michigan, University of Southern California, and others, Soar Technologies, Explore Reasoning Systems, Inc.	The SOAR project is a combined effort between several academic institutions and commercial ventures. It intended purpose is to bring intelligent behavior to simulation entities. It is built on top of the Tcl scripting language. (Rosenbloom 1994)
ModSim, ModSim II, SimProcess, SimScript CACI	Programming languages for sequential simulation. ModSim and ModSimII are based on Modula-2.

Table 1-5 Some commercially available 4th generation sequential simulation packages

While sequential simulation technologies are useful for small to moderately sized problems, there are problem scales where the additional computing resources available from distributed simulation become necessary. SODL, the language described in this dissertation, is in this category, along with two others: Yet Another Distributed Discrete Event Simulator (YADDES) (Priess 1990), and A Parallel Object-oriented SimulaTion LanguagE (APOSTLE) (Wonnacott 1996).

YADDES programs are translated into C and compiled using a standard C compiler. As such, much of the benefits of object-oriented programming are not realized in YADDES. It comes with both an optimistic and a conservative synchronization engine for handling messages and processing state changes. The fact that it can use either paradigm leads to some additional complexity in the language structure, namely that in order to realize any efficiencies in the conservative techniques, the process topology must be specified at compile time. It accomplishes this through an extensive specification of connections between topologically adjacent processes.

APOSTLE is similar in intent to YADDES, but differs by generating C++ code. This enables APOSTLE to be an object-oriented language, which it is. However, like YADDES, it is designed to work with both conservative and optimistic synchronization engines (though as of this writing, only the optimistic engine had actually been implemented). This again requires that the topology of the distributed simulation be specified prior to run time, by stating which outputs feed which inputs. One restriction on the APOSTLE system is that it only runs on Sparc platforms.

1.2.4. Distributed simulation standards

During the 1980's, the U.S. Department of Defense introduced the Distributed Interactive Simulation (DIS) standard allowing host processes performing a distributed simulation to communicate with each other using a common interface. Extensive libraries were developed for DIS, and it became a popular method of distributing simulation systems. Any DIS certified simulation system was able to send packets to or receive packets from any other DIS compliant system. This allowed different simulation systems to operate with each other, even though they had not originally been designed to do so. That is, simulation system X while DIS compliant may have been intended to work with simulation systems Y and Z. However, Z may not have been designed to operate with X. Thus, any messages passed from X to Z might be ignored when they arrive at their destination.

These interoperability issues led the DMSO to propose in 1995 the High Level Architecture (HLA). Here constructs called "federations" are established for each group of simulation system developers wishing to have their simulation systems interoperate (Kuhl 1999). These federations define standard object types and message formats allowing the different simulation systems within the federation to communicate with each other. DMSO has provided the HLA Run Time Infrastructure (RTI), an extensive set of communications routines, similar to CORBA. Actual distributed simulation implementations making use of HLA require developers to write a great deal of code to address the specific issues pertinent to their system's requirements. Apart from the format of the interface between components, a great deal of effort is also required to ensure compatible semantic content (i.e. ensuring that the same message means the same to all of the simulation components).

1.3. SODL system description

Neither YADDES nor APOSTLE makes assumptions about the underlying mechanism managing node synchronization in a distributed simulation. Since some methods[2] require a rigid pre-specification of the

[2] Conservative synchronization, described in more detail in Chapter 2 is one such mechanism.

communications topology for optimization purposes, this topology must be specified regardless of the synchronization method employed. This provides additional opportunities for programmers to introduce errors and complicates message passing in a purely dynamic fashion. More fundamentally, this specification requires modelers to think in terms that may not be appropriate for their specific application. For instance, some models might naturally require arbitrary interaction between virtual objects. This means that the communications topology would need to be fully connected, or some mechanism to forward messages needs to be introduced into the model.

This thesis introduces the Simulation Object Description Language (SODL) and takes a somewhat different approach. Like YADDES and APOSTLE, we designed SODL to facilitate distributed discrete event simulations (DDES). It does this by converting SODL source files into C++. It is therefore, like APOSTLE, an object oriented language, though perhaps not to the same extent. Where it primarily differs is in its assumption about the underlying simulation engine, namely that it will always be optimistic[3] in nature. Optimistic synchronization methods do not take into account communications pathways of a distributed simulation in any of its optimizations, freeing developers from specifying the distributed simulation system topology. Messages are simply addressed and delivered to the members in the recipient list. This notion makes SODL, a completely event driven language requiring inter-process communication to occur exclusively through message passing.

The guiding principle directing design decisions of the SODL system has been to make as clean a split as possible between the simulation engine and the behavior of the objects within the simulated environment. The behavior specification derives directly from the model description, and only rarely do simulation system restrictions interfere. Thus, SODL provides developers the freedom to express object behavior in terms that naturally arise from a model without having to be distracted with performing unnecessary run-time system declarations or calls to the underlying simulation engine to handle some action the engine could perform on its own.

SODL provides a framework upon which simulation system developers can simulate models that make extensive use of the notions of stimulus-response. That is, each of the objects in a distributed simulation system has a state and makes changes to that state based upon external stimuli. These stimuli can originate from any of the other objects in the simulated environment and need not flow over fixed communications pathways. SODL does retain the ability to optionally specify the communications pathways, but the decision to use this feature is solely at the discretion of the model maker and simulation system designer, and is not imposed upon them by constraints within the SODL language specification or its run-time system. Thus, most any model that can be framed in terms of stimulus-response can be directly coded into SODL source files from such an interaction specification.

There are some drawbacks to this approach. By removing the necessity of defining the topology, we lock ourselves into an optimistic approach, which is not always the best for a given application (Fujimoto 1993).

Another problem is that there is no convenient way for one object instance *A* to directly manipulate or access the data of another instance *B*. Instead, *A* must send a message to *B*, and it is *B*'s responsibility to manipulate its own data, or to reply to A's query about its internal state data. While this may seem awkward to code, it does more closely reflect the way things happen in the real world. That is, when objects interact in the physical world, they do so primarily by sending messages of one form or another. One person will speak to another. When an anti-armor round strikes a tank, it can be thought of as having sent the message "I just hit you" to the tank. Thus SODL's use of pure-event programming is, in the end, rather natural.

SODL draws a distinction between simulation objects (which SODL calls *processes*) and the data that is transferred between them (which SODL calls *messages*). Messages can have arbitrary data fields and methods to act upon them in a manner analogous to objects in traditional object oriented programming languages. The message with its payload is transferred between objects in a completely dynamic manner (meaning that no pre-specified topology is required to direct the message traffic). Processes, in addition to having internal data and methods, are able to send and receive messages. The process modifies its internal data upon receipt of a message.

[3] Optimistic simulation is described in more detail in Chapter 2.

SODL processes are also modal in nature; they can turn modes of operation on an off based on the message stream they receive. This allows a process to act one way upon receipt of a message at one time, and act completely differently upon receipt of another message with the same payload while in a different mode. This conveniently provides developers with the capability of radically changing a simulation object's behavior to a given stimulus (message receipt) with little difficulty. For instance, a simulation of an ant colony might have the ants behave in one fashion when they are searching for food, another when they actually find some and gather it, and still another when their nest comes under attack from a neighboring colony. A developer need only change modes when a certain condition is met, thereby fundamentally changing the object's behavior.

1.4. Scope

The primary purpose of this research was to provide a logical framework for defining object behavior in a virtual environment. To this end, we introduce a conceptual framework for discussing simulation, and upon this framework, define a language structure allowing simulation system developers to easily and quickly specify these object behaviors. When it became clear that a stimulus-response description could provide this specification, distributing the simulation across a network of computers seemed like a logical but secondary extension of the underlying work.

What we specifically avoid in our analysis are any measures of overall system performance. The rationale is that the current SODL run-time system can be modified to optimize its performance by taking advantage of new algorithms or techniques. We instead concentrate on the language specification itself and note that there is little in the way of programmer interface with the simulation system. This allows simulation system developers to concentrate on implementing a simulation of a model, rather than with the mundane, often error prone additional work other simulation systems require.

We provide in this document a description of the sequential simulation system, intended to simulate a distributed system and used to test SODL. In addition, a number of sample programs and associated descriptions are provided to gain some insight into the capabilities and limitations of the language specification and any run-time system that might eventually be employed to support it.

Chapter 2. Digital simulation

2.1. Overview

sim·u·late – *vt.* 1. To give a false appearance of; feign 2. To look or act like.[4]

Simulation has historically allowed scientists and analysts of various fields to test hypotheses about naturally occurring or hypothetical systems. For much of its history, simulation involved either a series of hand computations or analog devices designed to simulate some physical system. More recently, digital computers have allowed more sophisticated simulations in terms of their computational complexity, and been used in a wider variety of ways. Consider Bernoulli's description of airflow through a venturi. Prior to digital computers, aeronautical engineering relied heavily upon hand computations and wind tunnel testing (analog simulation). With the advent of digital computer technology, we now have the ability to cheaply and easily perform high fidelity digital simulations of airflow around an airframe.

While digital simulation systems are useful in describing physical systems that assist in analysis, other have been applied to training people to operate equipment that is either too expensive or too dangerous to actually train on. Examples of simulation for training include space flight operations, as well as nuclear power plant operations.

For digital simulation, we can create virtual environments with which people can interact more safely, and in some cases more cost effectively than the real-world systems. Yet, these virtual environments only exist as a collection of 1's and 0's in a computer's memory system and only reflect the real system in a way that is meaningful to those conducting the analysis or participating in the training.

The range of applications in which simulation might be useful is quite varied. As such, no one approach to simulation will be appropriate in all circumstances. In some instances, a few lines of equations scribbled on the back of an envelope might be sufficient for a particular purpose. In others, hundreds of digital computers working in concert with each other might only scratch the surface of some complex system dynamics.

[4] *Webster's New World Dictionary of the American Language*, David B. Guralink, Ed. 1979

For purposes here, we will be considering primarily simulations performed with a digital computer.

2.2. Modeling

Before actually getting to the point where a simulation is of any use, we quite often need to describe in some unambiguous manner the dynamics of the system under consideration. Modeling is this process of describing the system, and although it is not dealt with in any detail here, it is an integral part of the overall simulation process, and needs to be adequately tackled prior to writing any code. The modeling process in many cases will provide significant insights into a system's dynamics – insights that may actually obviate the need for simulation.

Before dealing directly with modeling, however, we need to provide some context. What follows is a framework around which we might construct some pertinent notions and to promote an understanding of some of the constraints inherent in digital simulation. While we make no claim as to whether or not any the following formalization of the modeling and simulation process appears in prior work, we developed and included it here because of its apparent absence in texts on the subject. Prior to formalizing this context, we provide a brief overview of these notions.

We start by introducing the notion of a *universe*. A universe may have multiple time dimensions (called the universe's *temporal component*). For each element in the temporal component, the universe has exactly one state. We normally are interested in universes with only a one-dimensional temporal component subject to some strict ordering. This induces an ordering on the universe's states, and allows us to impose a causal relationship between states.

Since a universe has a broader scope than we are normally interested in, we pare away much of the state information for a universe and concentrate upon one small portion of the universe, called a *system*. For the sake of discussion, let us consider the system of an object undergoing projectile motion under the influence of gravity. When considering this physical system we can ignore many of the minute influences that act upon this object, and look only at the very limited scope of the object and the primary gravitational sources acting upon it.

We then develop a *model* describing the system. Modeling formally describes how the system behaves. In the case of the object undergoing projectile motion, we look to Sir Isaac Newton's laws of motion to describe how the projectile moves. We note that the model is something that we humans have done to describe a physical process. It in no way dictates how the physical system really behaves. For instance, Newton's laws of motion are only an approximation of how objects really move. Finer predictions are possible with the introduction of Einstein's theories on Relativity. The physical system always behaved in a certain way regardless of what people say or think about it; it took Newton and Einstein to propose models describing this behavior.

Finally, we will want to study this model, to see how well it predicts physical systems, and perhaps learn new things about the system. We use simulation to perform this prediction. In the case of ballistic motion in a vacuum, we can simulate the behavior of physical systems quite easily with a pencil and paper. By employing a digital computer, we can incorporate other aspects of Newtonian motion (windage, or N-body interactions) to perform higher fidelity simulations in shorter amounts of time.

From here, we formally introduce the concepts outlined above.

Given an indexing set P and family of sets V_p, $p \in P$, let us define the Cartesian product ρ_P

$$\rho_P \equiv \prod_{p \in P} V_p \tag{2-1}$$

and the family of canonical projections (Hungerford 1974) π_p

$$\pi_p : \rho_P \to V_p \tag{2-2}$$

where $\pi_p(v)=v_p$, the p^{th} component of v.

From this, given a set of parameters P, we can define a universe U_P

$$U_P \subseteq \rho_P \equiv \prod_{p \in P} V_p \tag{2-3}$$

That is, for purposes here, the universe is simply a subset of the Cartesian product of the sets V_x. In order to maintain a degree of generality, we make no assumptions about the set P, or any V_p, $p \in P$, specifically, we make no claims as to their cardinality nor of the elements they may contain.

Let $T \subseteq P$, $P' = P - T$. We can redefine U_P in terms of P' and T by

$$U_P \subseteq \rho_{P'} \times \rho_T \tag{2-4}$$

We refer to T as a *temporal component* of P exactly when all of the following conditions are both satisfied:

U1. For all $t \in \rho_T$ there exists a unique $p(t) \in \rho_{P'}$ such that $(p(t), t) \in U_P$.

U2. $U_P = \bigcup_{t \in \rho_T} (p(t), t)$

When $T = \emptyset$ is the only temporal component of P for the universe U_P, then we say that U_P is a *static universe*. When U_P is not static, it is said to be a *dynamic universe*. Though in general there is no restriction on the set T, we normally think of universes having $T = \{\mathbf{R}\}$, where $\mathbf{R}$ is the set of real numbers, as their only temporal component. This provides a natural ordering of the states in the universe, and offers a convenient glimpse into how we might perform digital simulation – by calculating states in chronological order.

We will not usually be interested in considering the whole of U_P, but rather some subspace of it. Therefore, given a universe U_P, we define a *system* $S_R \times \rho_T$ over a collection of parameters $R \subseteq P'$ as

$$S_R \times \rho_T \subseteq \prod_{x \in R \cup T} V_x \tag{2-5}$$

We then define $r: \rho_T \rightarrow S_R$ such that

$$\pi_i(r(t)) = \pi_i(p(t)) \text{ for all } t \in \rho_T \text{ and } i \in R. \tag{2-6}$$

to ensure that the parameters in the system take on the same value as their associated parameters in the larger universe given the same time value.

From this, we can create a model in which we simplify the actual behavior by aggregating system parameters and create rules governing how these parameters interact.

First, we pick a finite set D to serve as an index for parameters in our *model* $M_D \times \rho_T$. We then choose a function $f{:}R \rightarrow D$ to aggregate all the system parameters into more manageable modeling parameters. So as not to overly complicate the model, we will impose the restriction on D and f that for all $y \in D$, there exists $x \in R$ such that $f(x)=y$. That is, f is surjective. If we were not to have this limitation, we could have a collection of model parameters that would need to be tracked, even though there is no analogous collection of parameters in the system S_R we are considering.

Next, we need to define another surjective function $g{:}\rho_R \rightarrow \rho_D$ aggregating the state of the system S_R into some state in the model M_D. The specific definition of this function, like f, is at the discretion of the modeler.

From this we get the definition of a model over a collection of modeling parameters D, $M_{D \times T}$ by

$$M_D \times \rho_T \subseteq \prod_{x \in D \cup T} W_x \tag{2-7}$$

and we define $d{:}\rho_T \rightarrow M_D$ such that

$$\pi_i(d(t)) \approx \pi_i(g(r(t))) \text{ for all } t \in \rho_T \text{ and } i \in D \qquad \textbf{(2}$$

(2-8) requires the model to approximate the system $S_R \times \rho_T$ to some problem specific degree and requires a great deal of discretion on the part of the individual defining the model to determine an acceptable error level.

The last part of the model is to describe how the various parameters interact with each other. This involves explicitly defining the family of functions $\pi_i(d(t))$ in a manner that will satisfy (2-8) to the desired degree.

An important consideration here is that models are closed, in that there is no influence upon the model parameters from a source other than other model parameters. Any such external dependence would become part of the model. There is no restriction on the system from which the model is derived. However, we are generally well advised to pick the system parameters wisely so that there is a reasonable expectation that no outside influence can significantly adjust any of the system parameters. We make no claim about the nature of the behavior of U_P or of how its parameters interact.

As mentioned earlier, the process of modeling a system through this abstraction is at least as important, and quite often just as informative, as actually performing a simulation. It can provide a great deal of insight into the underlying dynamics of the system that would have otherwise been unrealized. Picking the right parameters in the system and properly aggregating them is critical in developing an adequate understanding of the system under consideration. It is somewhat of an art form to create a model, given only raw data and observations; an art form we will explore no further in the confines of this publication. There is a great deal of material available for creating models and some assistance in this regard may be found in books such as (Fishwick 1995) and (Morrison 1991). Both have an extensive list of additional references that could be useful in specific modeling applications.

2.3. Digital simulation

From this point on, we will only be dealing with universes with the temporal component $T=\{\mathbf{R}\}$, the collection of real numbers. This imposes a complete order on the universe's states if we consider them in terms of the time at which each state occurs. When performing a simulation, we are concerned with causality. That is, a universe's state $p(t)$ can only be affected by earlier states $p(s)$ $s<t$. Simulation will seek to calculate the state of models for a collection of times of interest to the modeler.

Once we have a model of a system, the next task is to get a computer to tell us interesting things about the model we can project back to the system under consideration. The problem with digital computers, however, is that they are not very good at expressing with arbitrary precision state values we would like to consider in our model. They are also hamstrung by the fact that each operation a computer performs requires some non-zero time to compute. We formalize these constraints as follows:

1. Digital computers perform only finite precision arithmetic
2. Each operation on a digital computer takes some time, $t>0$, to perform

We note that by virtue of these restrictions, each simulation of a model must be broken down into a finite number of discrete, contiguous intervals. We will denote these intervals I_0, I_1, … I_{n-1} and define each by $I_i=[t_i, t_{i+1})$ with $t_0<t_1< \ldots <t_n$.

Condition 1 also requires us to limit our notion of the sets D and T. Dealing with T is a straightforward matter if we define $R^* = \{t_0, t_1, \ldots, t_{n-1}\}$, and $T^*=\{R^*\}$. D, on the other hand, is not as easily handled in a general and formal sense. Each set of model parameter values W_i, $i\in D$, must have a finite collection of associated approximations W_i^* that a digital computer can represent and manipulate.

We once again perform an abstraction, this time from $M_D \times \rho_T$ to the *digital simulation* $L_D \times \rho_{T^*}$.

$$L_D \times \rho_{T^*} \subseteq \prod_{x\in D} W_x^* \times \rho_{T^*} \qquad \textbf{(2-10)}$$

We go on to further describe the properties of $L_D \times R^*$ with $|t^*-t|<\varepsilon$, $\varepsilon>0$ and sufficiently small, we define $l:\rho_{T^*}\rightarrow L_D$

$$\pi_i(l(t^*)) \approx \pi_i(d(t)) \ \forall\ t^*\in \rho_{T^*},\ i\in D \qquad \textbf{(2-11)}$$

All of these abstractions and simplifications of the underlying system lead us inevitably to conclude that there can be a substantial difference between a real world system, and a digital simulation. This in no way mitigates the importance of simulation, but instead serves as a caution not to put too much credence in the output of one, especially if sufficient testing of the model and an associated simulation has not been performed. Specifically, there needs to be on the part of the modeler a rather deep understanding of the system being considered. This understanding needs to include an awareness of the degree of dependence upon initial conditions of the system (how chaotic the system is), and how closely the model tracks the system's behavior in reality.

To address these concerns, there is a Validation, Verification, and Authentication (VV&A) process within the United States Department of Defense whereby models and simulations are fine-tuned to more closely reflect the way the system actually works (DODI 1996).

Verification involves testing a simulation system to ensure that it reflects to an acceptable level of accuracy the specific model behavior for the system under consideration. That is, it verifies that the simulation system produces results that are consistent with the model it is supposed to simulate. Validation is the process of making sure that the model is a reasonably accurate representation of the system under consideration. This is normally accomplished after verifying the simulation by comparing results from the simulation with observations of the physical system.

Validation and Verification are two steps in an iterative process. A model is initially created to represent some system. It then is coded into some sort of digital simulation. Results from the simulation are compared to those predicted by the model, to ensure that it accurately reflects the intentions of the model makers. Once that is done, the simulation results are compared against real-world data to see if the model is an acceptable portrayal of the real world system. Refinements to the model are then made so that it more closely represents the system. These changes are then coded in the simulation, which must, in turn be verified again. This process is repeated until the simulation produces results within an acceptable tolerance of the real-world system.

Once the model and simulation are validated and verified respectively, an accreditation agent will certify that the model and simulation are fit for some specific use. Any enhancements to the model will require repeating the full VV&A process. Simulation changes require only verification and accreditation.

There are a number of techniques for actually performing the simulation on a computer. The techniques can be grouped into two main camps: Continuous Time Simulation (CTS) and Discrete Event Simulation (DES).

2.3.1. Continuous time simulation

Models requiring CTS approaches change their state in a continuous fashion and represent some continuous change in the system being analyzed. Such models quite often have as their rules a collection of partial differential equations governing parameter interaction. The field of numerical analysis is filled with numerical methods for simulating such equations. Euler's method is a simple approach that may be appropriate for some applications. Other systems may require more complex approaches such as Runge-Kutta. In any event, (Press 1992) provides a rich source of C code and some brief explanations for many of the most popular numerical methods for simulating systems of partial differential equations. (Atkinson 1989) is more theoretical in nature, not providing much in the way of source code, but providing helpful insights into some of the more common approaches. Finally, (Isaacson 1966) provides even more insight at the cost of being quite difficult to read.

A *continuous time model* is then formally defined as an $M_D \times \rho_T$ such that there exists $s_0 \in \rho_T$ such that for all $\varepsilon > 0$ there exists $s_\varepsilon \in (s_0 - \varepsilon, s_0 + \varepsilon)$ satisfying

$$d(s_0) \neq d(s_\varepsilon) \qquad \textbf{(2-12)}$$

As indicated above, even though the model state may continuously change with respect to time, the limitations of digital simulation require that it be broken up into suitably small time slices. Certain numerical techniques may change the size of these time slices, so we will make no specific assumptions about them. The goal of the simulation developer is when given the history $l(t_0)$, $l(t_1)$, ... , $l(t_{m-1})$ to determine $l(t_m)$ for $m \leq n$.

Since CTS systems are not the focus of this dissertation, we will not discuss them any further. More information on CTS systems is available in books such as (Hockney 1988), which offers a very extensive list of references that can be useful for specific applications.

2.3.2. Discrete event simulation

Models that can be thought of as changing in some fundamental way at only discrete instances of time are known as *discrete event models*. Though the system the model is meant to represent may be changing continuously, the model need not reflect that change. This is especially true for models of continuous processes, the partial differential equations of which can be exactly solved. Ballistic motion for instance can be solved exactly if certain simplifying assumptions can be made. Though an object undergoing ballistic motion is continuously changing its position, the parameters governing that motion are only changed at discrete instances of time. Thus, it can be modeled quite simply using a DES system.

Like CTS systems, DES systems are broken into discrete time slices. Unlike CTS systems, however, the model state is considered static between these iterations. Specifically, for all $t \in [t_m, t_{m+1})$, $0 \le m < n$ and for all $s \in [t_{p-1}, t_p)$, $0 < p \le n$ we have:

$$d(t_m) = d(t) \tag{2-13}$$

$$d(t_p) \neq d(s) \tag{2-14}$$

This translates well into the actual simulation, Equation 2-10, since we are forced to deal with things in discrete time slices by virtue of our restrictions.

Another major difference between CTS and DES systems is that the events needing to be processed in a DES system may not be generated in the order they are to actually be processed[5]. This has the potential to impact simulation system performance since sorting an unsorted list of objects has $\Omega(n \cdot \log n)$ time complexity. Practically speaking, however we might be able to do a little better. First, we note that we can never schedule an event to take place in the past, since that would violate causality. If we can further assume that the number of pending events does not exceed some constant value M (which is independent of n the total number of messages processed in a simulation run) then insertion into the pending event queue can be performed in $\Omega(\log M) = \Omega(1)$ time. Thus the overall time complexity for processing n events actually ends up being $\Omega(n \cdot \log M) = \Omega(n) \cdot \Omega(\log M) = \Omega(n)$. This is a reasonable simplifying assumption. If there is no such M then the number of pending events is not bounded, and will grow to fill the system

[5] This is perhaps the most important difference between continuous time and discrete event simulations. If discrete events are generated out of order, a discrete event simulation system is required to ensure that they are processed in the proper order.

memory eventually causing an abnormal termination of the simulation. In such cases, we will have to bound n.

2.3.3. Distributed discrete event simulation (DDES)

The focus of this dissertation is on distributed discrete event simulation. By this, we mean a discrete event simulation that is performed in a multiple instruction, single data (MISD) or a multiple instruction, multiple data (MIMD) environment (Fishwick 1995). Significant speedups can be achieved when additional processing power is applied to a simulation problem. The major complication in doing this stems from the fact that events need to be processed in the proper order, requiring a certain level of synchronization between the various nodes in the distributed simulation. This can be mitigated largely in MISD simulation topologies with some additional code to provide for proper synchronization. This is not nearly as straightforward in the case of MIMD topologies. It is not hard to imagine a circumstance whereby a message is delivered for processing to a node in a distributed simulation, only to discover that the node has already progressed beyond the intended processing time of the incoming event. Such errors are called *causality errors* (Fujimoto 1990).

At this point, we introduce notation common in most of the literature on distributed discrete event simulation, that being *Physical Processes* (PP) and *Logical Processes* (LP). If we further partition D into the N sets $D_0, D_1, \ldots, D_{N-1}$, we can induce a collection of physical processes PP_i^*, $i<N$, by

$$PP_i^* \subseteq \prod_{x \in D_i \cup T} W_x \tag{2-15}$$

with the following properties for each PP_i^*, $i<N$ and we can define $PP_i : U_T \rightarrow PP_i^*$ which satisfies

$$\pi_x(PP_i(t)) = \pi_x(d(t)) \text{ for all } t \in U_T,\ x \in D_i \tag{2-16}$$

This then induces the logical processes LP_i^* similarly by

$$LP_i^* \subseteq \prod_{x \in D_i \cup T^*} W_x^* \tag{2-17}$$

and we define $LP_i:\rho_{T^*}\rightarrow LP_i^*$ so that it satisfies

$$\pi_x(LP_i(t)) = \pi_x(l(t)) \text{ for all } t\in\rho_{T^*}, x\in D_i \qquad \textbf{(2-1)}$$

From this point, we will drop the * from LP_i^* and PP_i^* when referring to the logical and physical process. We will explicitly use $LP_i(t)$ and $PP_i(t)$ to refer to the states of LP_i and PP_i at time t, respectively.

Simulation then becomes defining or computing the collection of functions, $e_{i,j}(t_g, t_k)$, called *events*, that transform $LP_j(t_{k\text{-}1})$ to $LP_j(t_k)$ for $j<N$, $0<g<k\leq n$. This notation indicates $LP_i(t_g)$ scheduled the state change from $LP_j(t_{k\text{-}1})$ to $LP_j(t_k)$. These events can be thought of as messages transmitted between logical processes, containing enough information to allow receiving logical processes the opportunity to properly change their state. Thus, upon receipt of a message LP_i will change its internal state and issue additional output events.

DDES Approach	***Description***
Conservative	Causality errors are prevented from occurring, usually through some sort of blocking mechanism on each LP_i.
Optimistic	Causality errors are detected and, when they occur, the simulation system will recover from them.

Table 2-1 Distributed Discrete Event Simulation approaches

In distributed simulation, each LP_i could reside on different host processors, making communication via message passing a natural mechanism for inter-process communication. The problem becomes how to order events on each of these logical processes without creating causality errors. Alternatively, an approach at distributed simulation might allow causality errors to occur, but with sufficient care, a mechanism to detect and recover from them might instead be employed. These are the two main approaches used to ensure that the temporal integrity of each logical process remains intact. They are contrasted in Table 2-1.

Conservative techniques were the first to be adopted and employed in distributed simulation. There are a number of different algorithms available; two of the most popular conservative approaches are the Null Message Algorithm (NMA) (Chandy 1979), (Bryant 1977) and the Chandy-Misra Algorithm (CMA) (Chandy 1981). NMA prevents deadlocking states from being achieved while CMA has a mechanism to detect and recover from them[6]. These algorithms normally require the specification of a rigid communication topology whereby messages are sent from the outputs of one LP to the inputs of another through fixed channels. Knowledge of this topology – specifically, the topology's dependency graph – is critical in either avoiding or detecting and recovering from deadlocked states and for preventing causality errors.

Optimistic techniques have their origins in the notion of Virtual Time (Jefferson 1985a). Here all events and LPs have a time stamp that is used to maintain temporal consistency. Suppose an LP at time t_l receives a request to schedule an event at time $t_k<t_l$. The LP then must restore its state to time t_k. It must also revoke any events it issued after time t_k. The LP can then process all of the events it has for time t_k and later. The memory obsolete data occupies is periodically reclaimed in a process known as *fossil collection.*

The Time Warp Operating System (TWOS) (Jefferson 1987) was a research initiative in the late 1980's and early 1990's to investigate the performance improvement that could be realized through optimistic synchronization applied to distributed simulation. There is considerable literature available on the actual implementation (Reiher 1992, 1990c), debugging and optimization (Reiher 1990a, 1991b), and related topics (Reiher 1990b, 1991a).

The SODL system described herein makes use exclusively of optimistic synchronization based heavily on the approach in TWOS. While this may be problematic for some applications, (notably those with a high degree of coupling between logical processes) these are sufficiently extreme cases that their exclusion seemed a reasonable tradeoff, especially since other simulation languages (YADDES and APOSTLE, for instance) are designed to work with either conservative or optimistic approaches. The plus side of this tradeoff is that the simulation system developer is able to construct a much more loosely coupled simulation topology (since specification of communications channels are not necessary in optimistic simulation).

[6] A deadlock state in a distributed simulation is one whereby, due to a circular dependence of each of the logical processes, none of them can make any progress. It can be considered a generalization of the Catch 22 problem.

This then is context in which we conduct simulation. We have a universe, perhaps the one we all enjoy, or some hypothetical one, which has a host of systems within it. Some of these systems may exhibit behavior we would like to better understand. In the modeling process, we make formal behavioral descriptions of these interesting systems, based on the behavior we either conjecture or observe. We then simulate these systems to see if our model is an adequate representation of the system we are trying to understand, and make changes to more closely reflect it if it is not. Once the simulation and the model both adequately represent the system, we use the simulation to draw new conclusions, allowing us to better understand our world.

Chapter 3 discusses in more detail the notion of optimistic simulation, and describes in general terms how it is employed in the SODL run-time system.

Chapter 3. Overview of optimistic synchronization

3.1. Overview

The notion of optimistic synchronization came about in the mid 1980s (Jefferson 1982) in response to one of the criticisms of conservative methods. This criticism was that poorly balanced loads tended to render many LPs idle while they wait for slower LPs to complete their designated tasks, even when there may not be any specific dependency between the blocked LPs and those performing computations. While poor load balancing still adversely impacts optimistic synchronization, it does so only in cases where there is a data dependency between faster LPs and slower ones. Still, it is this one characterization that distinguishes conservative from optimistic synchronization; conservative synchronization attempts to avoid causality errors, while optimistic synchronization attempts to recover from them (Fujimoto 1993).

One of the problems with distributed simulation is that messages in transit between sender and receiver nodes are not always completely accounted for. These messages in transit can take an arbitrary amount of time to be delivered, and they may not necessarily be delivered in the order they were transmitted. They need to be accounted for in any distributed simulation algorithm.

In this chapter, we continue the analysis begun in Chapter 2, directed at the notion of optimistic simulation. Recall that a distributed simulation has a collection of logical processes LP_i, $i<N$, and a finite number of states $LP_i(t_d)$ for $t_d \in T^*$. Logical processes transition from $LP_i(t_{d-1})$ to $LP_i(t_d)$ in response to processing an event $e_{j,i}(t_g, t_d)$, which was generated on $LP_j(t_g)$. In this case, LP_i is the destination logical process, and LP_j is the source logical process of the event. The event processing time stamp, the virtual time at which the event is actually processed, is t_d. Finally, the event generation time stamp, the virtual time at which the event was actually generated, is t_g. We also will use the terms "event" and "message" more or less interchangeably throughout the remainder of this presentation.

We impose some restrictions on how the various LPs in the distributed simulation may behave:

1) In response to an event $e_{i,j}(t_g, t_d)$, $LP_j(t_{d-1})$ may change its internal state to $LP_j(t_d)$, and transmit a (possibly empty) collection of output events.

2) No LP may directly access the internal state data of another LP.

3) Events are time stamped. All events processed by a particular LP must be processed in time stamp order.

4) All events, $e_{i,j}(t_g, t_d)$, must be scheduled for some future time. That is $t_d > t_g$.

That in mind, we describe the basic Time Warp algorithm (Jefferson 82). We start by describing the various data structures we will need to facilitate node synchronization on each LP. Table 3-1 lists these data structures and describes how we will be using each of them.

We need to provide a mechanism for revoking messages that have been transmitted, in the event that this should become necessary. We therefore introduce the concept of an antimessage. Each antimessage $a_{i,j}(t_g, t_d)$ is associated with a particular event, $e_{i,j}(t_g, t_d)$. If LP_i receives an antimessage for an event, LP_i removes it from its event queue without processing it.

Data Structure	***Description***
event_p_queue(i)	This data structure is priority queue[7] that places the earliest event at the top. There should be some mechanism whereby no two messages have the same chronological value, despite having the same time stamp value. This can be accomplished by appending a unique ID field to act as a tiebreaker in the event of identical time stamp values. The next event to process is at the queue's top.
antimessage_p_queue(i)	This data structure is also priority queue that stores inbound event revocation requests. They are ordered in the same chronological order as their associated events, with the earliest antimessage always at the top of the queue.
state_queue(i)	This data structure, a traditional double-ended queue, retains copies of the state of LP_i for each event that is processed. Later states are at the back of the queue, while older states are at the front. The current state is normally inserted at the back, and older ones are removed from the front.
processed_event_queue(i)	This data structure, which can also be implemented as a double-ended queue, retains a copy of each of the events that LP_i processes. As each event is processed, it is inserted at the back of the queue. Older events are at the front of the queue.
output_event_queue(i)	This data structure can also be implemented as a double-ended queue. It retains a copy of all messages generated on LP_i, with the latest ones being pushed to the back of the queue, and the oldest in the front of the queue. They are ordered according to their generation time, t_g, not their delivery time, t_d.

Table 3-1 Data structures needed for implementing the Time Warp algorithm

[7] Priority queues are discussed in more detail in (Cormen 1990), pp149-150.

1) The state of each $LP_i(start\_time)$ is initialized. Bootstrapping events are also scheduled in each ***event_p_queue***(i). The remaining queues should be empty.

2) While there is an LP_i with at least one message to process or there are messages in transit:

3) Push the current state, $LP_i(t)$, into the back of ***state_queue***(i).

4) While the next message in ***antimessage_p_queue(i)*** revokes the next message in ***event_p_queue(i)***, remove and discard the top message of each priority queue.

5) Process the next event $e_{j,i}(t_g, t_d)$ in ***event_p_queue***(i), and push it into the back of ***processed_event_queue***(i). This results in setting the state of LP_i to $LP_i(t_d)$ and sending any outbound messages to the intended recipient LP's. A copy of each of these outbound messages is pushed into the back of the ***output_event_queue***(i).

6) Upon receipt to $LP_i(t)$ of the event $e_{j,i}(t_g, t_d)$, if $t_d > t$, it is inserted into the ***event_p_queue***(i) to be processed in chronological order with the other pending events. If $t_d < t$, then a rollback to time t_d is performed, and $e_{j,i}(t_g, t_d)$ is scheduled with the remaining events.

   a. To recover state $LP_i(t_d)$, pop from the back of ***state_queue***(i) until the back element has a time stamp $t < t_d$.

   b. Remove from the back of ***processed_event_queue***(i) each event $e_{j,i}(t_g', t_d')$ with time stamp $t_d' \geq t_d$, and reinsert it into ***event_p_queue***(i).

   c. Remove from the back of ***output_message_queue***(i) each event $e_{i,k}(t_g', t_d')$ that has generation time stamp $t_g' \geq t_d$, and send the associated antimessage $a_{i,k}(t_g', t_d')$ to LP_k.

7) Upon receipt to $LP_i(t)$ of the antimessage $a_{k,i}(t_g, t_d)$, if $t < t_d$, then insert $a_{k,i}(t_g, t_d)$ into ***antimessage_p_queue***(i). If $t \geq t_d$, then perform the rollback to time t_d described in 6 a-c above.

8) Periodically update the local estimate of the global virtual time, $GVTE_i$

   a. Pop from the front of ***state_queue***(i) all states prior to $GVTE_i$ except the latest one prior to $GVTE_i$.

   b. Pop from the front of ***processed_event_queue***(i) all events with delivery time stamp, $t_d < GVTE_i$.

   c. Pop from the front of ***output_event_queue***(i) all events with generation time stamp, $t_g < GVTE_i$.

Figure 3-1 Synopsis of the Time Warp algorithm

We also need to introduce here the concepts of Global and Local Virtual Time (GVT & LVT respectively).

Definition 3-1: Given a logical process LP_i, the *Local Virtual Time* for LP_i, $LVT_i(r)$ at real world time r is defined to be the time stamp, t_{di}, of the last event processed $e_{j,i}(t_{gi}, t_{di})$ at or prior to real world time r.

Definition 3-2: The *Global Virtual Time* at any real-world time r is defined as:

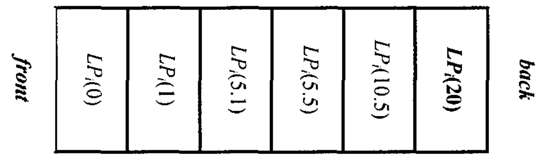
$$GVT(r) = \min\left(\bigcup_i LVT_i(r) \cup MT(r)\right) \quad \textbf{(3-1)}$$

Where $MT(r)$ is the set of message processing time stamps of messages in transit at real world time r.

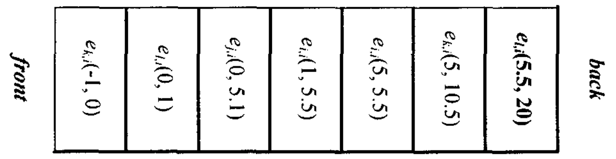
Here, $LVT_i(r)$ is defined as the local virtual time of LP_i at real world time r. Practically speaking, the GVT computation is fairly complex, though there are a number of algorithms available, notably (Bellenot 1990), (Fabbri 1999), and (Lin 1989). In all cases, the GVT value that is actually used approximates the real GVT. This is fine provided it does not overstate the actual GVT value.

We describe the basic Time Warp algorithm in Figure 3-1, using variables described in Table 3-1. There are a number of algorithms available to perform the $GVTE_i$ computation referenced in step 8. We discuss one such algorithm in section 3.5 below.

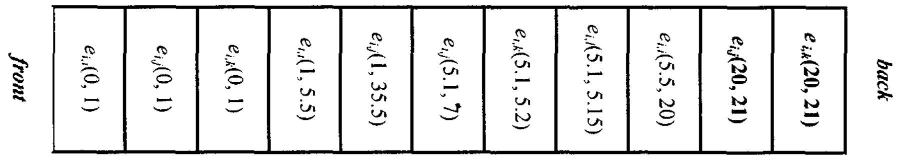
3.2. State saving



(a) ***state_queue***(*i*): $LP_i(t)$ saved states. $LP_i(20)$ is the current state



(b) ***processed_event_queue***(*i*)



(c) ***output_event_queue***(*i*): Each $e_{i,i}$, $e_{i,j}$, $e_{i,k}$, and $e_{i,l}$ will be delivered to LP_i, LP_j, LP_k, and LP_l respectively for processing at the proper time.

Figure 3-2 Saved state data of sample logical process at time 20

When LP_i receives a *straggler*, which is a message with a processing time stamp less than the LVT_j at the time of delivery (Fujimoto 1993), LP_i's state must be restored to a time that makes processing the straggler temporally consistent. Therefore, there is certain data needing to be saved in order to facilitate this state recovery. This includes the state of each LP_i, all processed events prompting state changes in LP_i, and any events LP_i generated because of processing earlier events (Reiher 1990b). Figure 3-2 depicts a typical implementation of the state saving process. This is somewhat different in the SODL run-time implementation, where several LPs are aggregated together into what is called an Engine. However, the same general idea is employed.

Figure 3-2 (a) shows the various $LP_i(t)$ values as the state of the logical process at time t. Each $e_{i,j}(t_g, t_d)$ is an event scheduled for LP_j generated at time stamp t_g and intended to be delivered at time stamp t_d. The t_g time stamp is only used by LP_i to facilitate the event revocation in the rollback mechanism. The t_d time stamp is used only by LP_j to properly order the event. The bold items in the figure 3-2 (a), (b), and (c) represent the components added to their respective double ended queues after processing $e_i(5.5, 20.0)$.

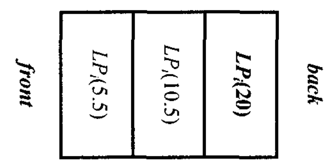
As events are processed and removed from ***event_p_queue***(i), they are stored in a ***processed_event_queue***(i). When an event for LP_i is processed for a time stamp later than the one that is currently at the back of ***state_queue***(i), a copy of the back element is added at the queue's back end, and the time stamp changed to reflect the event time stamp. The event is processed on the new back element, and any outgoing events are inserted at the back of ***output_event_queue***(i). The processed event is also inserted at the back of the ***processed_event_queue***(i) after the LP has completed processing the event.

3.3. Fossil collection

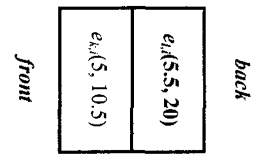
Since events can be scheduled to occur only in the future (per restriction 4 above) we can be assured there are no events processed before the Global Virtual Time (GVT).

Figure 3-3 reflects the data stored in the LP_i depicted in Figure 3-2 after receiving a notification that the GVT is not earlier than 7.5.

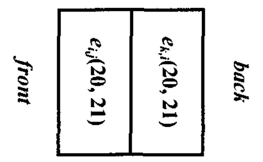
Upon notification of an update of the GVT, the process of reclaiming memory occupied by obsolete data can be performed. By restriction 4 above, no event can be sent into the past. Therefore, since the GVT is the lowest time stamp of all of the LPs in a distributed simulation, no pending events prior to the GVT remain. This fact allows us to reclaim most of the saved LP internal data with a time stamp prior to the GVT. Specifically, all members of ***processed_event_queue***(i), $e_{j,i}(t_g, t_d)$ where t_d<GVT can be removed from the front of the ***processed_event_queue***(i). This is easy to do since they were entered into the processed event from the back by their t_d value. Similarly, all members of ***output_event_queue***(i) generated on LP_i where t_g<GVT can likewise be reclaimed as no rollbacks can restore the LP to a state with time stamp prior to the GVT.



(a) ***state_queue***(i)



(b) ***processed_event_queue***(i)



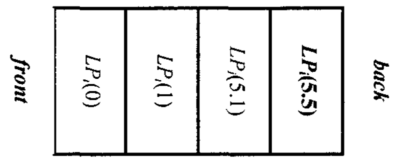
(c) ***output_event_queue***(i)

Figure 3-3 Result of fossil collection with GVT=7.5

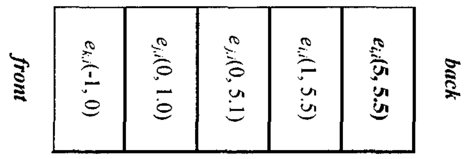
The story is slightly different for the saved state data in the ***state_queue***(i). Since the GVT is 7.5, we need to be able to recover state data for any time after 7.5. However, the earliest state we have after 7.5 has time stamp 10.5. This will do us no good if we need to recover state data for time 8.0, should that be necessary. The solution is to remove saved state data from ***state_queue***(i) up to, but not including the last time prior or equal to the GVT. Having done this, we can now recover the state to any point after 7.5. Since there can be no state changes in the LP for time stamp values in the range [5.5, 7.5], it will just be $LP_i(5.5)$.

This fossil collection process can serve a second purpose, other than just reclaiming memory. Specifically, during fossil collection, we can perform any irrevocable activity. Such activity could include writing data to a log file (or any IO activity for that matter) or allocating or deallocating memory not specifically related to the synchronization protocol.

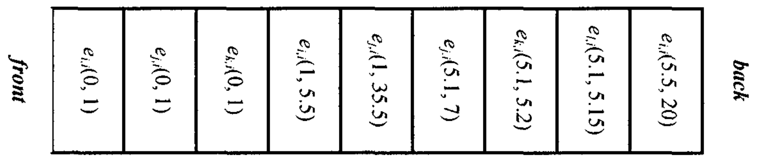
3.4. Rollback (state recovery)



(a) ***state_queue***(i) : $LP_i(t)$ saved states. $LP_i(5.5)$ is the current state



(b) ***processed_event_queue***(i): Events $e_{k,i}(5, 10.5)$ and $e_{l,i}(5.5, 20)$, which had been processed earlier were reinserted into ***event_p_queue***(i).



(c) ***output_event_queue***(i): Events $e_{i,j}(20, 21)$ and $e_{i,k}(20, 21)$ were revoked by sending antimessages $a_{i,j}(20, 21)$ and $a_{i,k}(20, 21)$ to LP_j and LP_k respectively.

Figure 3-4 Results of a state rollback on LP_i to time 6.0

Given $LP_i(t)$ and a newly received event $e_{k,i}(t_g, t_d)$ is called a straggler any time $t_d<t$ (Fujimoto 1993). Upon receipt of a straggler, $LP_i(t)$ must become $LP_i(t_d)$ in order to process the new event in a manner consistent with causality. Rollbacks can also occur when receiving an event revocation, $a_{j,i}(t_g, t_d)$ with $t_d<t$.

Figure 3-4 shows the effect of a rollback to time 6.0 from the state indicated in Figure 3-2.

Here, we see that all of the processed messages with $t_d \geq 6.0$ were popped from ***processed_event_queue***(i) and reinserted into ***event_p_queue***(i) for processing after the new event. All of the members of ***output_event_queue***(i) with $t_g>6.0$ were revoked and their associated antimessages were sent to annihilate them. The members of ***state_queue***(i) with time stamp $t>6.0$ are also removed and the memory they occupied is reclaimed. The new state, from which we can now process the new incoming message, has time stamp 5.5

We note that the state saving and rollback mechanisms used in the Time Warp algorithm keep all of the LP queues ***state_queue***(i), ***processed_event_queue***(i) and ***output_event_queue***(i) in chronological order according to their time stamps, t, t_d, and t_g respectively. This makes managing them straightforward, permitting all operations to take the form of either popping from or pushing onto the back or front of the respective queues.

3.5. Global Virtual Time computation

Though the Time Warp algorithm makes no specific mention of an algorithm for the Global Virtual Time computation, it is an integral part of the fossil collection process. Most algorithms used in GVT computation require that it be done synchronously. That is, all of the LPs need at the same point in real time to somehow communicate their current LVT with all of the other LPs. This can then be used to perform the fossil collection described above. This can impact system performance as each LP has to stop what it's doing, and wait for the computation to be completed. It then needs to perform the fossil collection. As a result, the system will periodically pause while all of this is going on. Some of the more popular synchronous GVT computations are (Bellenot 1990), (Fabbri 1999), (Lin 1989). (Bauer 1992) and (D'Souza 1994) both proposed somewhat different general asynchronous approaches to performing the GVT calculation, each with their drawbacks. (Fujimoto 1997) and (Xiao 1995) describe asynchronous methods specifically geared towards shared-memory multiprocessing systems.

We would like to make an observation about GVT, and try to relax requirements that might otherwise constrain various methods. We first show that the GVT increases monotonically given all events $e_{i,j}(t_g, t_d)$ in a distributed simulation, where $t_d > t_g$.

Theorem 3-3: let $x, y \in \mathbf{R}$, such that $x \leq y$. Then $GVT(x) \leq GVT(y)$. That is, GVT increases monotonically.

Proof: Suppose otherwise. Then there exists $x, y \in \mathbf{R}$, such that $x \leq y$ but $GVT(x) > GVT(y)$. Then at some time $x' \in [x, y)$ there was a rollback on some LP_k such that $LVT_k(x') < GVT(x)$, or an event $e_{k,j}(t_g, t_d)$ was generated on LP_k where $t_d<GVT(x)$. This second possibility can be dismissed quite simply by noting that such a case would violate the principle that a message have a processing time stamp strictly greater than its generation time stamp. In other words, it violates the requirement that $t_g<t_d$.

Let us therefore examine the first possibility in some detail. LP_k must have received from some LP_j an event $e_{j,k}(t_g, t_d)$ causing a rollback. Let us note that $t_g<t_d=LVT_k(x')<GVT(x)$ for this event. Now, either LP_j generated $e_{j,k}(t_g, t_d)$ before, at, or after real world time x. Let us consider each of these cases separately

I. $e_{j,k}(t_g, t_d)$ was generated before real world time x. Then it was delivered to LP_k after real world time x, meaning that it was a message in transit during the entire interval [x, x'). Specifically it was in transit at time x, implying that $t_d \in MT(x)$. From above we see $t_d<GVT(x)$, and we get $GVT(x)\leq\mathbf{min}(MT(x))\leq t_d<GVT(x)$ resulting in a contradiction.

II. $e_{j,k}(t_g, t_d)$ was generated at or after real world time x. Then we are forced to conclude that $GVT(x)>LVT_k(x')=t_d>t_g\geq LVT_j(x)\geq GVT(x)$, which is another contradiction.

Hence, the circumstance that $GVT(x)>GVT(y)$ cannot ever arise. ■

From here, we note that we can relax somewhat the requirements of the local estimate of the global virtual time on each LP_i without impacting the validity of the Time Warp algorithm.

Theorem 3-4: Given a collection of logical processes, LP_0, LP_1, ..., LP_{N-1}, each with local virtual times $LVT_0(r)$, $LVT_1(r)$, ..., $LVT_{N-1}(r)$, at some real time r, let $GVTE_i(r)$ be the local estimate of the $GVT(r)$ on LP_i at real world time r. Then no LP will ever have an unrecoverable causality error provided that $GVTE_i(r)\leq GVT(r)$.

Proof: Since $LVT_i(r)\geq GVT(r)$ for $i\in\{0, 1, \ldots N-1\}$, any $e_{i,k}(t_g, t_d)$ generated on LP_i at or after real world time r will have $t_d>t_g\geq LVT_i(r)\geq GVT(r)$. Now $e_{i,k}(t_g, t_d)$ will be delivered to LP_k at some real world time $r+\delta$. But, until delivery of $e_{i,k}(t_g,t_d)$ is actually performed, it is a message in transit and we get $t_d>t_g\geq GVT(r+\delta)\geq GVTE_k(r+\delta)$, allowing us to perform any necessary rollbacks on LP_k. ■

Corollary 3-5: If $GVTE_i(r)>GVT(r)$ for some i, r, then there is a possibility that an unrecoverable causality error may occur. Thus, the state of the distributed simulation in such a case is invalid.

Proof: Let i be such that $GVTE_i(r)>GVT(r)$. Since $GVT(r)$ is the minimum of all the local virtual times and all messages in transit, there is either a j such that $LVT_j(r)<GVTE_i(r)$ or a message in transit with time stamp $GVT(r)$. It is possible that either LP_j, or a message in transit with time stamp less than $GVTE_i(r)$ causes (ether directly or indirectly) a roll back to a time prior to $GVTE_i(r)$. ■

An important upshot of Theorem 3-4 is that as long as no $GVTE_i(r)$ overstates the actual $GVT(r)$ no two $GVTE_i(r)$ values need to be the same.

One might be tempted to use these results to develop a token-passing asynchronous GVTE computation. For instance, consider an algorithm whereby a token is passed around a ring of nodes in a distributed simulation system. This token contains a payload allowing the receiving LP_i to compute $GVTE_i(r)$. It then adjusts the payload of the token, and passes it along to $LP_{(i+1) \bmod N}$. Each LP_i will have different estimates of the GVT, and at first glance, it would appear that the hypothesis of Theorem 3-4 is satisfied. This is not the case.

Consider the situation depicted in Figure 3-5. In this case, the $GVTE_i(r)$ is based upon the state of LP's at real world times earlier than r. It is possible that in the intervening time, some LP_j could have had a rollback to some time $LVT_j(r)<GVTE_i(r)$, meaning that $GVT(r)<GVTE_i(r)$, opening the possibility of an unrecoverable causality error.

$LVT_0(r_{i(0)})$
$LVT_1(r_{i(1)})$
$LVT_2(r_{i(2)})$

(a) – LVT values used to compute $GVTE_2(r_{i(2)})$

$LVT_0(r_{i(2)})$
$LVT_1(r_{i(2)})$
$LVT_2(r_{i(2)})$

(b) – Actual LVT values at time $r_{i(2)}$.

Figure 3-5 Possible unrecoverable causality errors in asynchronous token passing GVTE calculation

This problem is not limited only to a token ring approach, but any GVTE computation that uses data based upon obsolete LVT values. This explains in part the popularity in synchronous approaches such as (Bellenot 1990) and (Lin 1989). The problem with most synchronous approaches to GVT computation is that they require processing to stop on all the nodes in the distributed simulation while the GVTE computation is performed. This has the possibility of adversely impacting performance of the simulation system.

Several asynchronous algorithms have been suggested, notably (Concepcion 1990), (Bauer 1992) and (Mattern 1993). We will focus particular attention her upon one of these approached.

(Mattern 1993) suggested coloring LPs either white or red. A white LP sends white messages, and a red LP sends red messages. All LPs are initially white. The approach is essentially to count the number of white (red) message that have been sent and received while the color of all the LPs is red (white). Once the difference between the sent and received messages is zero, a lower bound of the GVT can be calculated by getting the minimum time stamp that occurred during the collection of white (red) messages.

Mattern provided a formal description of his algorithm as it specifically applied to ring topologies, but provide no formal correctness proof. We suggest a more general version of Mattern's formal algorithm, making no assumption about the simulation topology, and prove its correctness.

Table 3-2 lists the various routines that we use in the general algorithm, as well as a description of their function. Table 3-3 lists the various data structures and a description of the data they contain. Finally, Figure 3-6 lists a generalized variation on the original ring-topology algorithm Mattern proposed.

Routine	***Description***
Global_Min(x_i)	A distributed procedure to return the global minimum of all the values, x_i for $i<N$. This value is returned to each LP_i at the completion of the call.
Global_Sum(x_i)	A distributed procedure returning the sum of all the values, x_i for $i<N$. This value is returned to each LP_i at the completion of the call.
Synchronize()	A distributed procedure to force all of the nodes in the distributed system to start at the same point in the GVTE computation at approximately the same real world time
Reset_Min$_i$()	Sets a local variable, tm_i to the current LVT_i on the host making the call.
Get_Min$_i$()	Returns the minimum LVT_i value that occurred since the last ***Reset_Min***() call. This minimum is adjusted if necessary every time a rollback occurs on LP_i.

Table 3-2 Routines used in the asynchronous GVTE computation

Data Structure	***Description***
$p_i \in$ {**red, white**}	This "color" is toggled between the two possible values between successive iterations of the algorithm. Each event $e_{k,i}(t_g, t_d)$ takes on the color p_k at the real world time it was generated.
$sent_i[p_i]$	The number of messages sent from LP_i with color p_i initialized to <0.0, 0.0>.
$received_i[p_i]$	The number of messages received by LP_i with color p_i initialized to <0.0, 0.0>.

Table 3-3 Data Structures used in the asynchronous GVTE computation

GVTE_Computation() is run on a separate execution thread on each LP_i during the entire distributed simulation run. This allows the main simulation engine to continue processing events during the GVTE computation. Actual implementation can make some modifications to the parameters of the various loops allowing processing to continue in the main portion of the simulation engine without occupying too much time in this routine.

```
GVTE_Computation()
    int outstanding                 // Number of outstanding messages with phase $p_i$
    int old_$p_i$                   // Current value of $p_i$, prior to being incremented

    Synchronize();                  // Make certain all are doing this for the same $p_i$
    while(true)
        old_$p_i \leftarrow p_i$        // Pre-incremented value of $p_i$ is used in the computation
        Reset_Min$_i$()             // Get the current simulation time.
        $p_i \leftarrow (p_i+1)$ mod 2  // Pass messages with this new $p_i$.
        do
            outstanding $\leftarrow$ Global_Sum(sent$_i$[old_$p_i$] - received$_i$[old_$p_i$])  // Count them
        while (outstanding > 0)             // Until all messages w/ phase old_$p_i$ are received
        $GVTE_i \leftarrow$ Global_Min(Get_Min$_i$())// Get the current global virtual time estimate
```

Figure 3-6 Asynchronous GVTE algorithm

The main portion of the simulation engine increments $sent_i[p_i]$ and $received_i[p_i]$ as it sends and receives messages with color p_i, respectively. At the start of the main loop, we store the current simulation time, retain the current value of p_i in old_p_i, and increment the color, p_i. In the inner loop, the variable *outstanding* will be non-zero until all messages with color old_p_i are received. Any rollbacks on LP_i to a time less than the minimum value retained at the ***Reset_Min***() call adjust that minimum to the new, lower LVT_i value.

Theorem 3-6: During application of the algorithm in Figure 3-6, $GVTE_i(r) \leq GVT(r)$ for all r during which the algorithm is in use.

Proof: When all of the messages with color old_p_i are eventually received we have $\textbf{Get\_Min}_i() \leq LVT_i(r)$ for all i, leading to $GVTE_i(r) = \textbf{min}(\textbf{Get\_Min}_i()) \leq \textbf{min}(LVT_i(r))$.

We now need to show that the $\mathbf{min}(\boldsymbol{Get\_Min_i}())\leq\mathbf{min}(MT(r))$. Since all remaining messages in transit $e_{i,j}(t_g,\ t_d)$ have color p_i, they have $t_d>t_g\geq\boldsymbol{Get\_Min_i}()$. It follows, therefore, that $\mathbf{min}(\boldsymbol{Get\_Min_i}())\leq\mathbf{min}(MT(r))$.

Thus $min(\boldsymbol{Get\_Min_i}())\leq\mathbf{min}(LVT_i(r)\cup MT(r))=GVT(r)$, satisfying the hypothesis of Theorem 3-4 and ensuring that the algorithm is correct. ■

If message acknowledgement is used as part of the communications protocol, the above algorithm can be slightly modified to explicitly wait until all messages with color old_p_i have been acknowledged. This modification would be in lieu of the summing of the $sent_i[]$ and $received_i[]$ arrays.

Chapter 4. SODL Run-Time System Architecture

4.1. Overview

Discrete event simulations require that processes change their state in accordance with some sequence of chronologically ordered events. In the SODL system, these changes are invoked because of receiving a message. Each message has a time stamp dictating when in the simulation run it is to be processed. The SODL run-time system is built to simulate a distributed simulation system to demonstrate that the language can be used in conjunction with optimistic synchronization mechanisms.

The purpose of the SODL simulation run-time system is to ensure that messages are delivered in the proper order to the proper simulation process. It does this through a modified version of the Time Warp algorithm discussed in Chapter 3. Whereas the basic Time Warp algorithm aggregates a great deal of behavior into a logical process, the implementation of the SODL run-time system disaggregates portions of the algorithm to provide certain economies of scale, improved granularity control, and sequential mode testing.

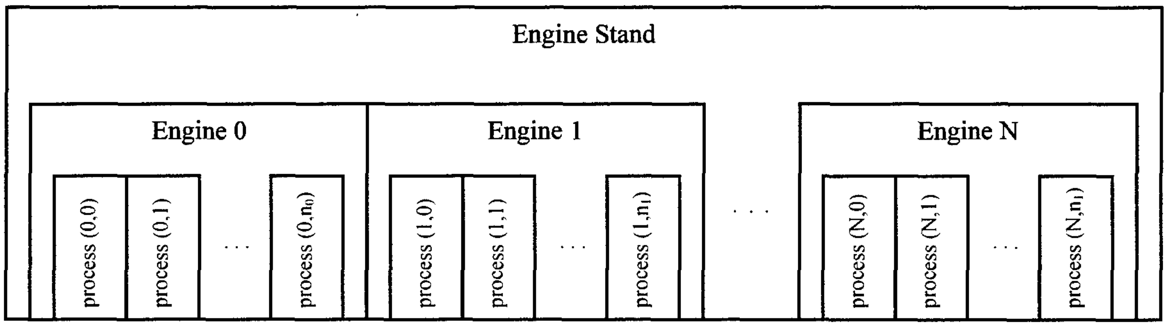


Figure 4-1 SODL system hierarchy

SODL has two types of user-defined objects, called constructs. Message constructs allow process constructs to interact with each other. There are a number of other objects in the SODL run-time system provided with the SODL distribution package. Industrious end-users may change or completely rewrite this run-time system to meet their particular needs. The overall architecture of the SODL run time system can be thought of in hierarchical terms. There is an engine stand, which can be thought of as distributed simulation system for purposes of testing the SODL system. This stand contains one or more simulation engines, each of which can be thought of as a node in a distributed simulation system. These engines act independently of each other, in a manner similar to nodes in a distributed simulation. Each engine has a number of processes it controls. This hierarchy is depicted in Figure 4-1.

4.2. Message constructs

Messages provide the means for objects within a simulated environment to communicate with each other. SODL messages have a designated type, which may be derived from another message type (ala object oriented inheritance). Figure 4-2 depicts the structure of message constructs.

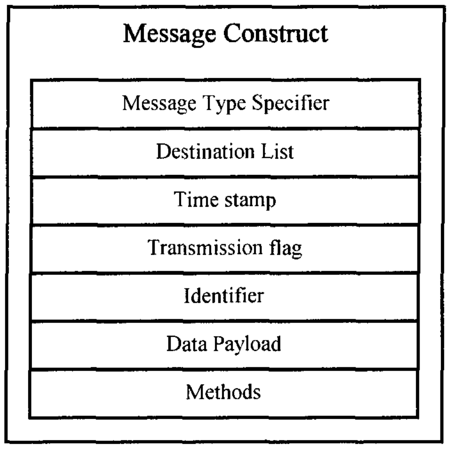


Figure 4-2 SODL message construct

4.2.1. Message Type Specifier

Each message has an associated type. This type determines how processes receiving the message will react to it. Since SODL aspires to be an object oriented programming language to some degree, messages can inherit portions of their functionality from parent messages constructs. Unlike some other languages, (most notably C++) only single inheritance is allowed. This design consideration was made primarily to simplify implementation. Messages of type B, derived (either directly or indirectly) from some message type A, are said to be of type A and B. This abstraction allows additional flexibility in message delivery and processing. Routines provided in the run-time system can make use of these relationships.

4.2.2. Message destination list

The destination list is determined at run time and need not be the same for any two messages. Each destination has a unique identifier that acts as an associative address for delivering the message. Users can establish a default recipient list at compile time for a given message. They can, alternatively override or augment the default recipients at run time.

4.2.3. Message time stamp

In order to ensure that messages are processed in the proper order, each message has a time stamp. Messages with earlier time stamps will be processed before those with later time stamps. In cases where two messages with the same time stamp value are encountered, the message identifier is used as a tiebreaker. This allows all messages generated to fall into a unique ordering, regardless of the order generated.

4.2.4. Message transmission flag

The SODL language requires that all possible outgoing messages be declared prior to compiling the source code files. In certain cases, it may not be desirable to actually transmit all of the messages that could be transmitted in response to an incoming message. The SODL run-time system provides programmers with a mechanism to preempt message transmission. They can set the message transmission flag to **false** to accomplish this end. The message transmission flag is set by default to **true** and must be either changed directly or by overloading the function called to examine the message delivery flag. See Chapter 6 for more details.

4.2.5. Message identifier

Each message has a unique identifier that allows it to be tracked down in the event of a revocation, and to provide a complete ordering in the event that two messages have the same time stamp value. This identifier has two components. The first is the index of the engine instance (see section 4.5 below) where the message was initially generated. The second is the actual instance count of the message generated on that engine.

4.2.6. Message data payload

The simulation developer specifies the data payload at compile time. This payload is analogous to the data members in a traditional object oriented programming language.

4.2.7. Message methods

Methods are analogous to the methods found in object oriented programming languages. They are intended to act on the data members of the particular message instance to which they are associated.

4.3. Process constructs

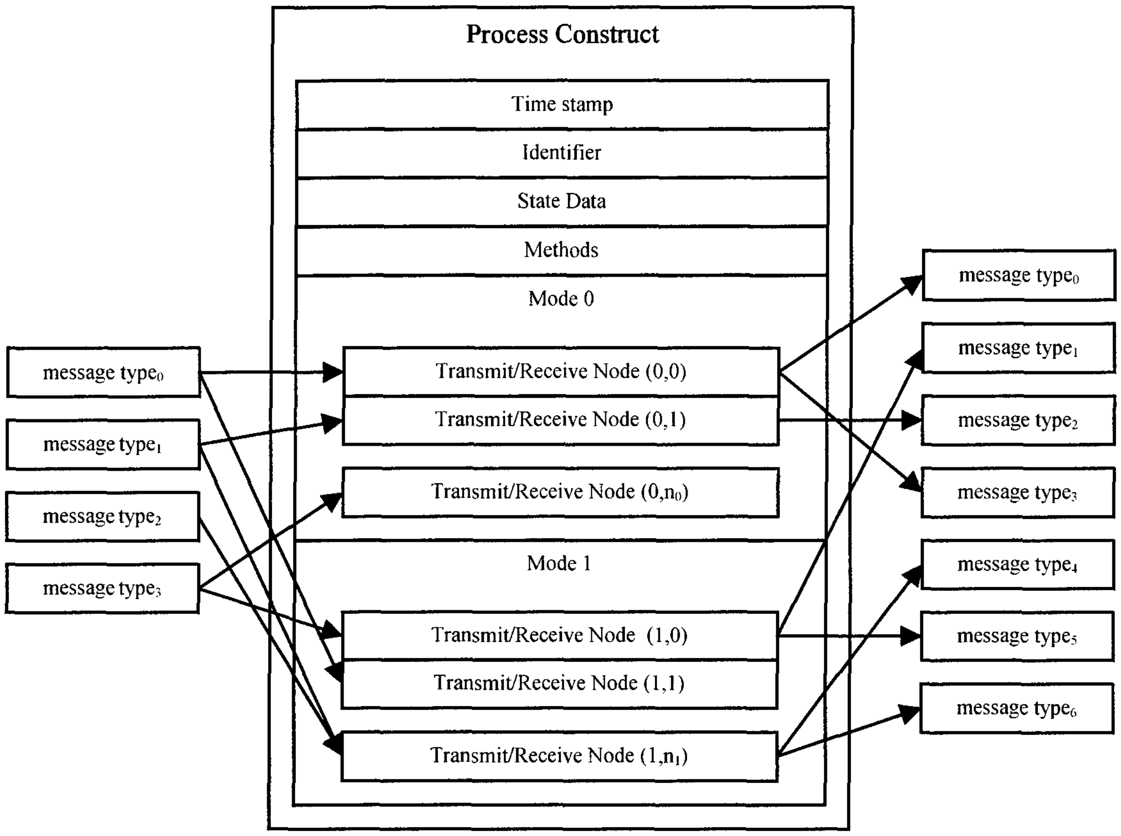


Figure 4-3 SODL process construct

From the programmer's perspective, all of the functionality associated with a logical is encapsulated into a SODL process construct. Each process can send messages and change its internal state upon the receipt of a message. There are enhancements allowing certain tasks to be performed somewhat easier, namely the notion of a process mode. Figure 4-3 shows the basic structure of a SODL process. This particular example shows a process with two modes, but in can in general have any number of them. Each mode has a collection of transmit/receive nodes. Each of these nodes accepts one message of a stated type (which implicitly includes all derived types), changes the internal state data when it receives a message, and produces output messages. There is no linguistic limit to the number of modes, the number of nodes in each mode, nor to the number of output messages a node can transmit[8].

It might seem a little odd that node (1, n_1) receives both messages of type 1 and 2., while node (0, 1) can only handle messages of type 1. Messages are declared as types that can inherit data and methods from a parent message. If message type 1 is a parent message of type 2, and if node (1, n_1) is intended to receive messages of type 1, then technically, any message of type 2 is also a message of type 1, and node (1, n_1) can process it.

4.3.1. Process time stamp

Each process instance has a time stamp associated with it. This time stamp is changed to the time of the message that is currently being processed. The process controller (see Section 4.4) uses the process time stamp to facilitate state saving and recovery.

4.3.2. Process identifier

Each process has associated with it a unique identifier. This identifier is a pair of numbers that correspond to the process's owning SODL engine (See section 4.5) and an index for distinguishing between all of the processes the parent engine controls.

4.3.3. Process state data

State data is analogous to the member data found in an object oriented class definition. This state data has an associated time stamp. The process is said to have the state of its data elements at the time of its time stamp. Changes to the state data are considered instantaneous. This leads to the situation where a state may not be completely up to date if messages with the process's current time stamp are still pending. Though the order of message delivery is the same from run to run of the simulation, for practical purposes developers should not rely on any particular processing order for messages with identical time stamps.

State data should not contain references, since the C++ standard has trouble copying objects with them if the copy constructor is not explicitly defined. Pointers can be used, but doing so should be done carefully.

[8] Any limitations stem from the architecture and capabilities of the machine running the simulation.

Since the pointer value is copied in a copy constructor, and not the data that is being pointed to, great care must be taken to ensure that the state of the data being pointed to is consistent, and can be rolled back if it is dynamic in nature. SODL provides callbacks allowing developers to perform some processing in the event of a rollback and during fossil collection so that, among other things, data in a pointer may be corrected and processed if necessary.

4.3.4. Process methods

Methods can be used to perform calculations or modify internal process state data. The SODL engine hides references to other processes, so these methods cannot generally be called on other process instances, even instance of the same type.

4.3.5. Process modes

Modes can be thought of as a collection of transmit/receive nodes (described in section 4.3.6) and can be activated or deactivated independently of each other. Only nodes in active modes can receive messages. At startup, all of the modes are active. Each process receives a bootstrapping message that can be used to deactivate modes that are intended to be dormant at the start of the simulation. Alternatively, prior to actually sending a message to a process, an initialization method is called in each process instance that can also be used to deactivate desired modes. This is discussed in more detail in Chapter 6.

4.3.6. Process nodes

Each mode can have a collection of subordinate nodes. Each of these nodes handles exactly one type of input message, including any messages derived of that type. The node is directly responsible for processing the message, ensuring that proper updates to the process state are made, and that the data fields in any outgoing messages contain the proper values. Input messages are passed into the node by value, instead of by reference. Output messages are passed by reference. The reason for this is that input messages may be used by other processes, or by the very process currently handling the message. This also hides data fields from derived message types. Output messages are passed by reference so that they may retain any changes to them the node may make.

4.3.7. Process inheritance

Like messages, process can singly inherit behavior from parent process constructs. There are some intricacies associated with this practice that make this somewhat more complicated than inheritance in the traditional object-oriented sense. Inheritance of the process methods and data members is like that in C++.

Modes and nodes are a little different, though. Specifically, modes with the same name across an inheritance are actually the same mode instance. That is, if process A had a mode M, and a process B derived from process A also has a mode M, then these are the same mode in each instance. M can be activated or deactivated within the context of either A or B, affecting its activity state in both contexts.

Nodes also need to have their behavior defined more clearly, since there is no analogous feature in other object-oriented programming languages. Mainly, any nodes in active modes can have process messages of their input type. Overloading a node in a derived process construct will not prevent the message from being delivered to the parent's context. For instance, in the example above, assume that M has a node named N in both processes A and B. In this case node N in both A and B will process the message. The programmer does not explicitly pass the message to the parent class; the run-time system will do this implicitly. The reason for this is that, even though the name of the node may be the same, the output messages may differ, and for ease of implementation, this approach was implemented.

4.3.8. Fossil collection in the process instance

When the process controller performs fossil collection, it is safe to produce any output that may be pending for the state at its designated time stamp. SODL provides the ability for developers to overload the ***fossilCollect*** method; all output should be performed in this method. The remaining SODL run-time system ensures that fossil collection occurs in the proper process order, so that output appears in its proper sequence as well.

4.4. Process Controllers

A process controller manages many of the Time Warp specific functions associated with each process. Process controllers ensure that the state is saved prior to processing messages that will change the time stamp of the process it controls, and when directed to do so, performs fossil collection and rollbacks. It also acts as a conduit for sending messages to and from its process. Figure 4-4 shows the general structure of a process controller and how it receives and transmits messages to and from the process.

4.4.1. Identifier

Each process has a unique identifier, which we discussed in section 4.3. The process controller has an identifier with a little additional information, including typing data that makes possible screening inbound messages from only specific types of messages.

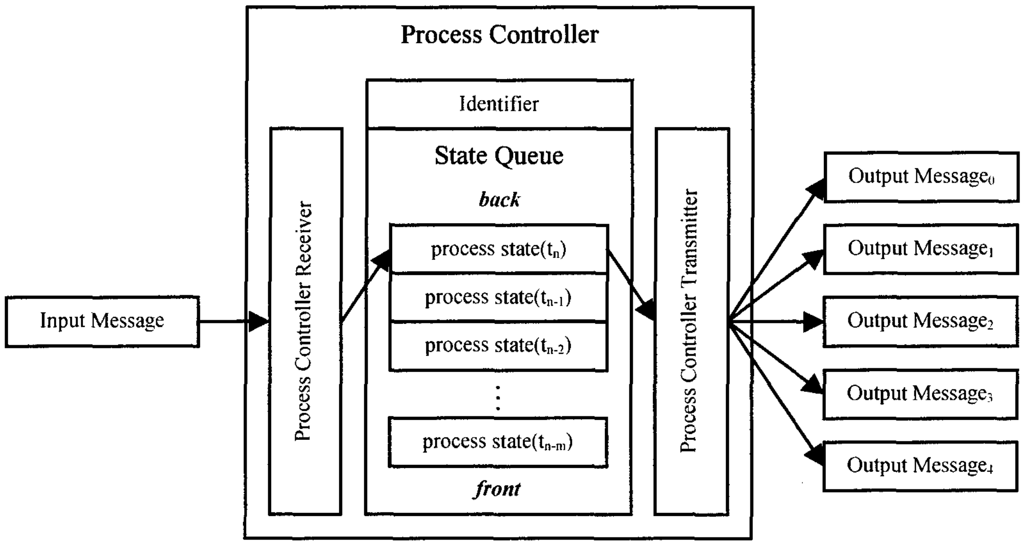


Figure 4-4 Process Controller message flow

4.4.2. State queue

The state queue holds the process states for specific points in time. The back element of the state queue is called the current state. Due to the Time Warp algorithm as implemented in the SODL system, the process states are in descending order of their time stamp value from the current state at the back and earlier states toward the front. States are saved onto the back of the queue, and they can be removed either from the back of the queue (in the case of a rollback) or from the front (during fossil collection).

4.4.3. Process controller message receiver

Upon receipt of a message delivery to the process controller, the time stamp of that message is compared with that of the back element. There are three cases, with which to contend:

I. **Time stamp of incoming message < Time stamp of current state**: It should not normally happen that a message is received with an earlier time stamp than the current state.

II. **Time stamp of incoming message > Time stamp of current state**: Create a new current state by copying the old one onto the back of the state queue and changing its time stamp value to that of the incoming message. The new back element is now the new current state. The process controller also registers a fossil collection event with the controlling engine so that this new state can be reclaimed later. We now deal with the message as if it has the same time stamp as the current state, in case III below.

III. **Time stamp of incoming message = Time stamp of current state**: The message is passed to the current state, which can make modifications to its internal data and generate outgoing messages.

4.4.4. Process controller message transmitter

Requests for message transmission originate in the process instance associated with the controller. Output messages are preprocessed and screened. The process controller will examine each output message and reject transmission of any that have their transmission flag set to false or any that have an empty destination list. Also, since we requires output messages from a node to have a greater time stamp value than their current time stamp, each outgoing message time stamp is set to a value slightly greater than the current time stamp if this condition is not satisfied.

4.4.5. Rollback

The SODL engine managing the process controller may periodically direct a rollback to a time *t*. All states saved in the state queue that have a time stamp not earlier than *t* are removed from the state queue. Since they have been placed into the queue in ascending order of their time stamps, this is simply a matter or removing the back element from the queue until the queue's back element has a time stamp strictly less than *t*. There is no easy way to revoke the fossil collection scheduled for this rollback, so when we are notified that one must take place for any rolled back states, the request is ignored.

4.4.6. Fossil collection

Some intricacies associated with fossil collection bear mentioning. First, in order to perform a rollback to a time *t*, a state with time stamp prior to time *t* must remain. That is, we must always have a state remaining that is prior to the current GVT. Secondly, since we are performing operations such as output during the fossil collection phase, we need to ensure that we also perform this output in time stamp order. These considerations led to an arrangement whereby fossil collection was conducted in two phases. Each fossil collection event the process controller performs has an associated time stamp, which is the time stamp of the process state that is obliged to perform some form of output. Any states with earlier time stamps have their memory reclaimed, but the one that performs the output is retained until the next cycle of fossil collection for that process controller. This is depicted in figure 4-5.

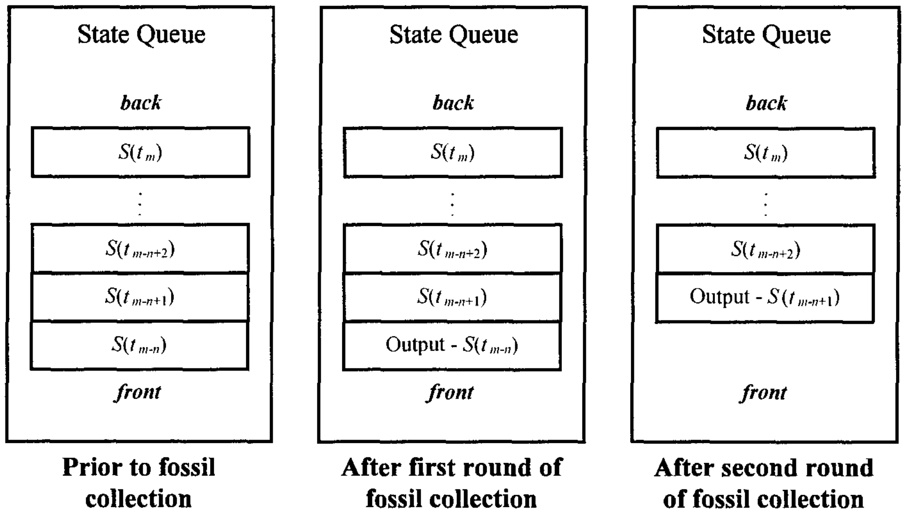


Figure 4-5 Fossil collection cycle in a SODL process controller

This approach allows the fossil collection to be conducted in a manner consistent with SODL system requirements. By performing the output at the designated fossil collection time, we guarantee that the output is produced in the proper time stamp order. By retaining the state that had just produced the output until the next fossil collection round, we provide for the possibility of rolling back to that state.

4.5. Engines

Engines aggregate multiple processes and provide some improvements in memory management and granularity control over the way a simulation run. All messages addressed to a particular process are passed first to the engine controlling that process and placed in an event queue for scheduling. The engines also retain ***sodl::AntiMessage*** instances for all messages that have been produced in subordinate processes so that those messages can be rolled back in the event that becomes necessary. The engine structure is depicted in Figure 4-6. All processes in an engine are considered to have the same time stamp value, though in practice this need not occur. This time stamp value is that of the current message being processed or, if no message is being processed, the time stamp value of the last processed message. Upon receipt of a message from another engine with a time stamp that is less than the current engine time stamp, a rollback to the new message time stamp is required of all subordinate process controllers.

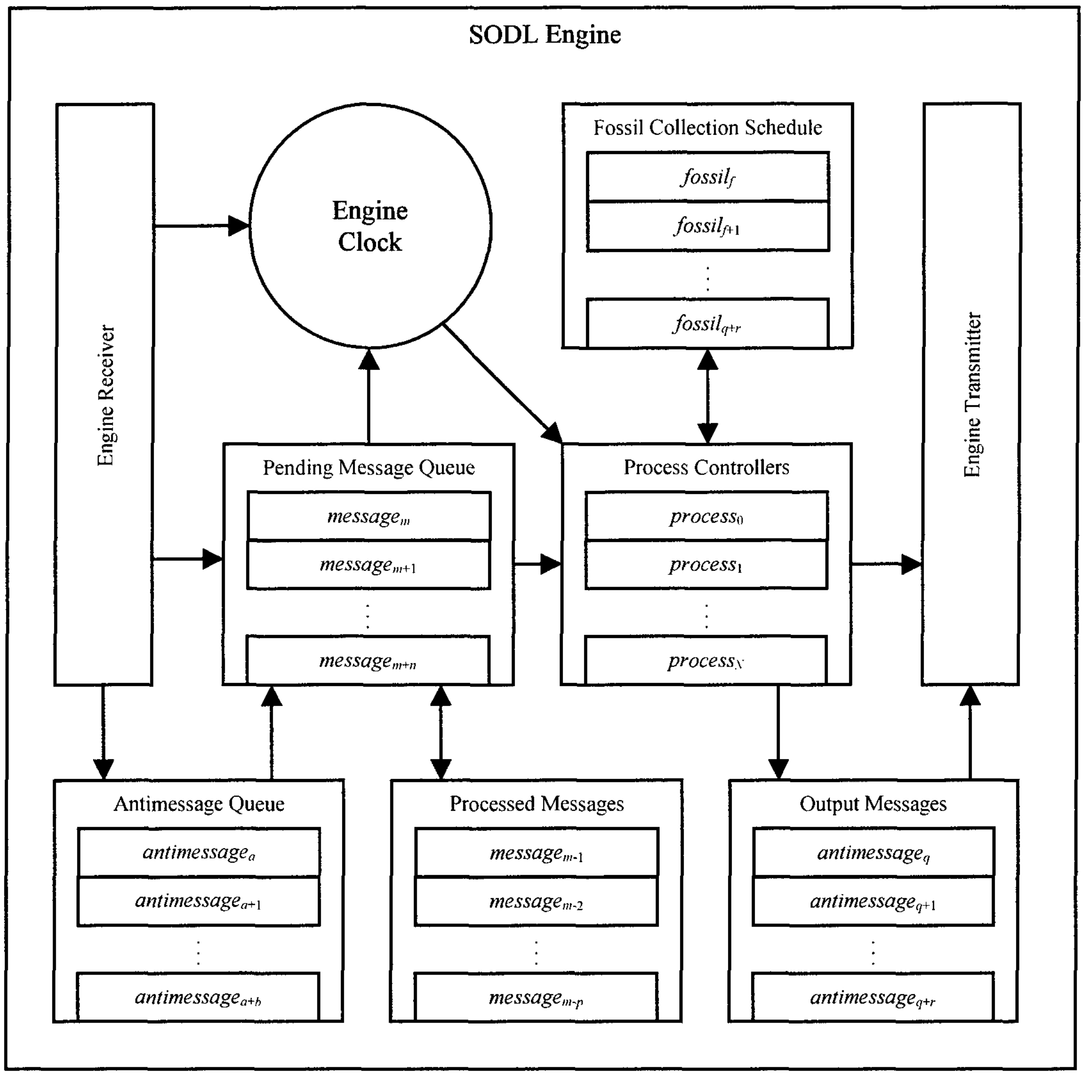


Figure 4-6 SODL engine structure

Each engine has an associated node number. This node number is unique among all of the other engines that may be in a SODL system. This number corresponds to the first part of the identifier for processes controlled on the engine, and of messages originating on the engine.

4.5.1. Local clock

The clock provides a convenient central way of checking the current simulation engine time stamp, and determining the time stamp of outgoing messages, should the user-defined portion fail to provide an adequate value. Earlier implementations of the clock had a real-time mode that allowed messages processing to occur at some rate proportional to the real world flow of time. This was found to be an unnecessary feature, though the general capability remains if developers wish to restore this capability.

4.5.2. Pending message queue (event queue)

The pending message queue prioritizes pending messages so that the next message in the queue has the lowest time stamp value of any others in the queue. For that reason, this is implemented as a standard library priority_queue (Josuttis 1999).

4.5.3. Antimessage queue

The antimessage queue stores antimessages are associated with messages in the pending message queue. The order restrictions on the two queues are identical, so if a message has been revoked, it can be checked with the top element of the antimessage queue when it is considered for delivery to its destinations. If the antimessage annihilates the message, both are removed from their respective queues and destroyed.

4.5.4. Processed message queue

Each message is inserted into the back of the processed message queue after the engine has processed it. In the event of a rollback, elements from the back of the processed message queue can be removed and reinserted into the pending message queue as necessary. During fossil collection, messages can be reclaimed from the front of the message queue. The messages in the processed message queue are ordered by their time stamp value.

4.5.5. Output message queue

During message transmission, a copy of the outgoing message's associated antimessage is retained in the event that a rollback is necessary. These messages, unlike those in the processed message queue, are not ordered by their time stamp values, but by the time stamp of the process state that created them (i.e. their creation time). The reason for this distinction is that during a rollback, any messages that need to be revoked are done so because the process that created them became invalid. We are not interested in the delivery time, but in revoking them because since they never should have been created in the first place. Thus, they are inserted into the output message queue in the order of their creation. During rollback, the antimessages are removed from the back of the queue and transmitted. During fossil collection, they are removed from the front and their memory reclaimed.

4.5.6. Process controller array

Each engine has a collection of processes it owns. Each of these processes has its process controller. The engine does not actually have any direct manipulation of the processes themselves, but can interact with the process controllers. Pointers to these controllers are stored in a standard library vector. Each process can be uniquely addressed by a pair of numbers, the index of its owning engine, and the index in the vector that has the pointer to the process controller. This is in fact the basis for the identifier in the process controllers and their processes.

4.5.7. Fossil collection schedule

The engine needs to keep track of any new states that have been created by the owned process controllers so that when fossil collection occurs, those states can be reclaimed in a chronologically correct sequence, ensuring proper formatting of the output. A fossil collection schedule is implemented as a priority_queue that has the earliest scheduled fossil collection event at the top.

4.5.8. Engine message receiver

Messages destined for a process controlled on some engine must first be passed to the engine's receiver. If the incoming message has a time stamp t_m less than the current clock time stamp t_c, then the engine initiates a rollback to the message time stamp. If the incoming message is an antimessage, it is inserted into the antimessage queue; otherwise it is placed in the pending message queue.

4.5.9. Engine message transmitter

Any of the process controllers owned by an engine can request a message to be transmitted. The engine does not consider the destination of the message at this point, but blindly forwards the message to the engine stand (see section 4.6 below) for the local node in the distributed simulation for delivery to the proper engines. An antimessage for the outgoing message is retained in the event a rollback requires its revocation.

4.5.10. Engine advancement

Periodically, the engine stand will instruct each engine it controls to process a message. When this occurs, the engine removes from the top of the pending message queue the first message that does not have an antimessage waiting for it in the antimessage queue. All message-antimessage pairs are removed and eliminated. The message is then sent to the controllers for all of the destination processes the engine owns. Once that is completed, the message is then inserted in the back of the processed message queue in case it needs to be reinserted into the event queue due to a rollback.

4.5.11. Rollback

An engine rollback is performed any time it receives an incoming message with a time stamp t_r less than the time of the local clock. When this occurs, each process controller the engine owns performs its rollback, as described in section 4.4.5. The engine must also rollback portions of its data structure as well. This is accomplished by reinserting into the pending message queue all of the messages in the processed message queue with time stamps less than or equal to t_r. Any antimessages in the output message queue with time stamps less than or equal to t_r are transmitted so that their associated messages can be revoked as well.

The fossil collection schedule remains unchanged. It is difficult and time consuming to weed out fossil collection events made irrelevant because of the rollback. When they are processed, the process controller can easily recognize that they have been the result of a rollback, and they are ignored.

4.5.12. Fossil collection

Fossil collection is broken into two phases, known as incremental fossil collection and gross fossil collection.

4.5.12.1. Incremental fossil collection

Incremental fossil collection is geared primarily towards the process controllers. The engine stand during the overall fossil collection process will query the engine as to the time of the event in the fossil collection schedule. When certain conditions are satisfied (see Section 4.6) the engine will be allowed to perform an incremental fossil collection, allowing the process state with the lowest time stamp value remaining to be fossil collected (see section 4.4). This will ensure that from the engine's perspective all of the output governed by the engine is generated in the proper order.

4.5.12.2. Gross fossil collection

Gross fossil collection takes place after incremental fossil collection at the direction of the engine stand, and is for reclaiming all of the engine's data with a time stamp value less than some t_f. Any messages in the processed message queue or the output message queue with time stamps earlier than t_f are removed from the backs of their respective queues, and their resources are reclaimed.

4.6. Engine Stand

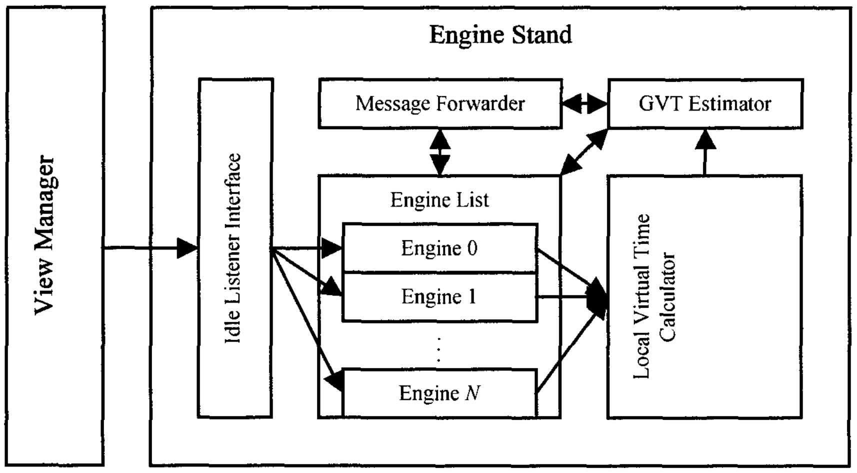


Figure 4-7 Engine stand structure

Each node in the distributed simulation has a unique engine stand, depicted in Figure 4-7, which acts as the primary controller for all of the engines managed on that node. It was introduced primarily as a means to implement the Time Warp algorithm without introducing problems associated with actually distributing the system. It proved useful in this regard in tracking down errors within the implementation of Time Warp in the SODL system.

It is retained because its value does extend beyond simply debugging purposes in SODL system development. Specifically, it provides a mechanism for testing and optimizing possible distributions of processes across engines, while keeping at bay network errors that might occur specifically in a fully distributed implementation.

4.6.1. Idle listener interface

The idle listener interface provides a mechanism for the view manager (see section 4.8) to control the engine stand. This control comes in two forms. The first is a request to perform any initialization required to get the simulation correctly configured for startup. This can include establishing initial bootstrapping messages, and process state initialization. The second form of control allows the engine stand to progress in the simulation. This is actually implemented by allowing each of the engines under control of the engine stand to advance. The view manager is notified if no pending messages remain so that it can end the simulation run, if that is its behavior.

4.6.2. Engine List

The engine list contains a reference to all of the engines in the simulation. Only certain engines are actually controlled by the engine stand that owns it in a distributed simulation. This provides an easy method of making certain that process instantiation is done consistently across the distributed simulation.

4.6.3. Message forwarder

The message forwarder sends and receives messages between engine instances. All messages are forwarded from the simulation engine to the ***sodl::EngineStand::stand*** instance. From there, the distribution list is queried, and copies of the message are sent to each of the ***sodl::Engine*** instances with processes listed as recipients.

4.6.4. Local virtual time (LVT) calculator

The LVT calculator keeps track of messages that have been processed and acknowledgements of messages transmitted to other engine stands. It then keeps track of the local virtual time for them engine stand, as defined in Chapter 3.

4.6.5. Global virtual time (GVT) estimator

The GVT estimator receives periodic requests for input into a global virtual time calculation. The GVT calculator gets the current LVT from the LVT calculator, and passes that on to the message forwarder to be provided for the GVT computation. At the end of the GVT computation, each engine stand receives the newly estimated GVT and passes it from the message forwarder to the GVT estimator. The GVT estimator updates the local estimate of the GVT, and conducts fossil collection.

4.6.6. Fossil collection

As in the engines, engine stand fossil collection is conducted in two phases. Upon an update of the local estimate of the GVT, incremental fossil collection is performed, followed by gross fossil collection.

4.6.6.1. Incremental fossil collection

The incremental fossil collection involves polling all of the locally controlled engines for their next scheduled fossil collection event. These are sorted and processed in time stamp order up to the GVT. Upon completion of an incremental fossil collection event on engine *e*, the engine stand again polls *e* for its next fossil collection event. In this way, all of the fossil collection events prior to the GVT are performed in proper time stamp order on all the engines the stand controls.

4.6.6.2. Gross fossil collection

After completion of the incremental fossil collection up to the new local estimate of the GVT, each engine the stand controls is given the opportunity to perform gross fossil collection to reclaim memory occupied by obsolete data directly under the control of the engine instance.

4.7. Message Passing Interface (MPI)

The SODL run-time system was intended to work with the Message Passing Interface (MPI), a standard library linkable with C, C++, and Fortran programs (Gropp 1998, 1999a, 199b). It is primarily used in general distributed programming, not specifically distributed simulation. However, certain features make it a useful tool in distributed simulation:

- It has a standard to which all implementations must adhere. It has also been widely used for other purposes, meaning that most implementations are reasonably mature and stable.
- It remotely starts up all of the nodes in the distributed simulation with the need for the simulation operator to do this manually.
- It has been ported to multiple platforms. In particular, it can operate heterogeneously with a variety of Unix platforms. It has also been ported to Microsoft Windows NT, but it does not interoperate with MPI on Unix platforms.
- Since the library has been ported to multiple platforms SODL run-time systems using MPI can easily be ported to those platforms with little or no code changes.

This aspect of the simulation system was not implemented prior to this writing, though it is hoped that derived work will establish a fully distributed implementation of the SODL run-time system using MPI for network communication.

4.8. View manager

The View Manager is a configurable subsystem that is designed to facilitate graphical output from a standard API. It controls exactly one ***sodl::IdleListener*** instance (of which ***sodl::EngineStand*** is a subtype). View managers have a start method that, when called starts the simulation running. This includes initializing the idle listener (which in turn initializes all of the simulation components) and incrementally stepping the simulation (by processing some non-zero number of pending events). SODL comes with two view managers, though developers can easily write new ones to work with API's not currently supported.

4.8.1. Text view manager

The Text view manager provides no graphics support. It is intended for producing output only to stdout or log files. Upon starting the view manager, it immediately calls the initialization routine in the idle listener.

It then calls the idle method in the idle listener until no messages remain in the simulation system to be processed. When that happens, the text view manager returns control to main.

4.8.2. GLUT view manager

The GLUT view manager provides a basic interface with the GL User Toolkit (GLUT) API. During initialization, any simulation objects that own a ***gvm::View*** instance registers it with the controlling GLUT view manager. Any user input events are then forwarded to the appropriate view. Multiple views can be added to the GLUT view manager, and it will ensure that the user inputs are sent to the proper display controller. GLUT provides a mechanism whereby during idle times in the graphics subsystem, a callback can be made to a static method. This mechanism is used to allow the simulation to process some pending events.

The GLUT view manager requires extensive additional support in the form of SODL processes and messages in order for developers to make use of it. This interface is described in more detail in Chapter 9. To summarize, there is a SODL process associated with each graphics object under the management of a GLUT view manager. These views may be distributed across a network, or they may be consolidated on one host machine. Inside each view is a scene graph that corresponds to the hierarchy of graphics processes in the distributed simulation. Messages can be sent to these SODL processes causing some state change in the receiving process. These changes are then forwarded to all of the views that have elements in their scene graph associated with the process. Upon receipt of these messages, the view generates a message and places it into a queue for processing during the fossil collection phase. It is only at the fossil collection phase that these changes to the scene graph are actually committed.

Chapter 5. SODL Parser Usage

5.1. Overview

The SODL parser is a software tool that translates SODL construct files into a collection of C++ source code files. It also creates a makefile for compiling the generated C++ source code. This makefile is intended for use with GNU make 3.79. Generated C++ source code files can be compiled using the GNU C++ Compiler (GCC version 2.95.2).

5.1.1. Cautionary notes

Programmers developing under Win32 operating systems will need to obtain a copy of the GNU make utility (version 3.79 or later). They are strongly advised to obtain the latest version of Cygwin (available at http://sources.redhat.com/cygwin/) and use that as a development environment. I personally recommend Emacs for Win32 (which is available at http://www.gnu.org/software/emacs/windows/).

In the event that a distributed simulation system is eventually produced with MPI, SODL will be restricted to running in a distributed mode under Unix systems until a robust version of MPI is produced to work in the Cygwin environment; as of this writing, there is no such implementation.

5.2. Installation

The parser is distributed as a tarred and bzipped source code with a variety of makefiles that can be used to build the parser for a variety of platforms. The makefile may be edited to direct the executable build to a specific location if the default settings are not satisfactory. Currently the following platforms are supported (though if I've been a good programmer, others should be equally well supported without major changes to the parser source code):

Platform (Compiler)	***Makefile name***	***Build command line***
Cygwin (GCC)	src/Makefile.Cygwin	make cyg
Linux (GCC)	src/Makefile.linux	make lnx
Solaris (GCC)	src/Makefile.solaris	make sol

Table 5-1 Methods for making the SODL parser

Installation and compilation instructions are in Figure 5-1.

1) Download the latest version of the software and copy it to a desired location.

2) `tar -xvjf sodl-x.x.xxx.tar.bz2` where x.x.xxx is the version number of the SODL distribution. If your version of tar does not support this form of decompression, use

```
bunzip2 sodl-x.x.xxx.tar.bz2
tar -xvf sodl-x.x.xxx.tar
```

3) `cd sodl-x.x.xxx`

4) `make` *platform-abbreviation*

5) `make clean`

Figure 5-1 SODL Parser (sp) installation instructions

5.3. Directory Structure

There are some makefiles available to perform various tasks for managing the contents of the directory structure.

Makefile command	*Function*
Make build	Builds SODL parser and sample programs; the default platform is Cygwin.
Make clean	Removes garbage files created during the build process.
Make fullclean	Removes garbage files and executables created during the build process.

Table 5-2 Makefile commands

Upon extraction, the there will be a number of directories along with the make files to build the executable.

5.3.1. ./bin

The executable is placed in this location after it is built. Add this to your path or move sp to a place in your path.

5.3.2. ./config

Contains configuration files for various platform and option combinations. The platforms are those listed above, and the options involve the view manager and whether or not the simulation is to run in a distributed mode.

5.3.3. ./doc

Some HTML-formatted documentation is available here. It is largely portions of this dissertation converted to HTML for portability.

5.3.4. ./object

Object files generated during the build process are places here. They can conveniently be removed by using the "make clean" after building is complete.

5.3.5. ./sample

Directory	***Contents***
./sample/*xxx*/bin	Location of the binary executable after the build is complete
./sample/*xxx*/build	Location of the C++ files generated by the SODL parser
./sample/*xxx*/object	Location of the object files generated by the compiler during the build process
./sample/*xxx*/plan	Location of the SODL source files which are used to generate the simulation

Table 5-3 Sample simulation system directory structure

Make command line	***What it builds***
make	all samples
make glut	battle, bounce1, bounce2, brigade1, hierarchy
make text	brigade2, ping, ring1, ring2, simple1, simple2, simple3
make dist	relay1, relay2, relay3, relay4, relay5, relay6
make battle	battle
make bounce1	bounce1
make bounce2	bounce2
make brigade1	brigade1
make brigade2	brigade2
make ping	ping
make relay1	relay1
make relay2	relay2
make relay3	relay3
make relay4	relay4
make relay5	relay5
make relay6	relay6
make ring1	ring1
make ring2	ring2
make simple1	simple1
make simple2	simple2
make simple3	simple3

Table 5-4 Make command line arguments for building demonstrations

The sample directory contains a number of SODL sample programs and a collection configuration files for various platform/option combinations. It includes the makefiles needed to manage the subdirectory contents. Each demonstration directory has a number of subdirectories that are used to build the samples.

These directories are described in table 5-3. Table 5-4 describes the shows the command line make arguments for build some or all of the demonstrations.

The next few sections provide brief descriptions of each of the demonstrations. They are more fully described in Chapter 9.

5.3.5.1. ./sample/battle

The battle demonstration is an autonomous tank battle simulator. Two opposing forces each with 25 tanks and 1 command post start out in some initial configuration and attempt to destroy the opposing force's command post.

5.3.5.2. ./sample/bounce1

The bounce1 demo simulates a collisionless system of particles in a closed container. It uses the GLUT view manager to display the simulation state.

5.3.5.3. ./sample/bounce2

This appears essentially the same thing as bounce1 above, but it is done with fewer messages

5.3.5.4. ./sample/brigade1

This is another GLUT view manager demonstrator that shows the progress of a military brigade performing some task. It also performs a great deal of output to stdout indicating which components are doing what.

5.3.5.5. ./sample/brigade2

This simulates the same thing as the brigade1 demonstration above, but without the GLUT view manager. It produces only textual output.

5.3.5.6. ./sample/hierarchy

Hierarchy is a single process GLUT view manager demonstration. Its notion is somewhat similar to the brigade demonstrations, but it does things in an apparently more orderly manner.

5.3.5.7. ./sample/ping

Ping simulates a token being bounced between two processes. It produces only textual output.

5.3.5.8. ./sample/relay1

Relay sets up a multi-engine simulation with two processes. One transmits a token to the other, which is then repeatedly bounced between them until the user stops the simulation. It was intended to act as a simple test of multiple engines without the possibility of a rollback ever occurring.

5.3.5.9. ./sample/relay2

This simulation is intended as a stress test of the rollback mechanism in the SODL run time system. For every message delivered, two are generated, so this simulation will eventually run out of memory and cause an abnormal termination. Each process resides on different engines and, upon receipt of a message transmits two messages, one to itself, the other to the partner process. Each of these messages has a random time stamp, which may cause a rollback to occur on the other.

5.3.5.10. ./sample/relay3

This simulation system is also intended to stress test the rollback mechanism and memory management of the SODL system. One controller process owns 1000 subordinate processes distributed 10 each on 100 engines. At startup, the controlling process sends a message to all of the subordinate processes. Upon receipt of such a message, each of these subordinates sends a message to a random subordinate at a random time.

5.3.5.11. ./sample/relay4

This simulation consists of a controller process, two subscription processes, and four child processes. The child processes each reside on their own engine. Each child subscribes to one of the two subscriptions processes. Messages sent to a subscription process are forwarded to all of its subscribers. The simulation starts when the controller sends a message to each of the subscriptions. That message is then forwarded to all of the children processes. Upon receipt of a message from the subscription, each child process sends a message to a random subscription, a message to unsubscribe from a random subscription, and a message to subscribe to a random subscription. Each of these messages has a random time stamp. Like relay2 above, this normally will create more messages each cycle than are consumed, and it therefore will eventually terminate abnormally due to lack of memory.

5.3.5.12. ./sample/relay5

Relay5 has three processes, a source, a relay, and a sink. Each process resides on a different engine. Messages periodically originate in the source and are sent to the relay. The relay forwards a message to the sink, which sends no messages.

5.3.5.13. ./sample/relay6

Relay6 is another test of the rollback mechanism. There are two processes on different engines, one process makes fast progress, and the other makes slower progress but at specific points in time, sends a message to the faster. This causes rollbacks to occur at predictable points in simulation time.

5.3.5.14. ./sample/ring1

This demonstration has a controller and ten ring elements arranged in a ring topology. Upon receipt of a message, each ring member transmits a message to the next member in the ring. The simulation is started when the controller sends a message to the first element in the ring. The ring topology is actually glued together with a subscription similar to that described in relay4.

5.3.5.15. ./sample/ring2

This behaves much the same way as ring1 above, except that when started, the controller broadcasts a message to all of the child processes. This results in each process processing messages in parallel (from a virtual time perspective) rather than sequentially as in ring1.

5.3.5.16. ./sample/simple1

This is a simple test of a single process that sends a message to itself upon receipt of one. It does this 100 times before stopping.

5.3.5.17. ./sample/simple2

This behaves like simple1, except that it does not stop. It was used to check for memory leaks in the main simulation engine.

5.3.5.18. ./sample/simple3

Simple3 behaves like the ping demonstration mentioned earlier, except that the messages between the two processes stop only when the user terminates the program.

5.3.6. ./src

This directory contains the source code required to build the SODL parser.

5.3.7. ./template

This directory contains the source code for the SODL run-time system. This includes a number of SODL construct files for the GLUT view manager, and, in the ./template/gvm subdirectory, the actual graphics engine that the GLUT view manager uses for graphics display.

5.4. Command Line Options

Once the parser is installed, you can run it by entering '**sp**' followed by a collection of flag values and the root process name.

sp [-a*baseDir*] [-b*binSubdir*] [-c*cfgDir*] [-d*display*] [-i*bldSubdir*] [-l*platform*] [-m*mode*] [-o*objSubdir*] [-p*plnSubdir*] [-t*tmpltDir*] [-v] *RootProcess*

Option	*Meaning [Default Value]*
-a	Specify base directory for the others below [./]
-b	Specify the binary subdirectory, where the executable will be generated [bin/]
-c	Specify configuration file subdirectory [$(SODLHOME)/config/]
-d	Specify either 'Text' or 'GLUT' display type [Text]
-i	Specify intermediate build subdirectory [build/]
-l	Specify platform name [*platform* used in building the parser]
-m	Specify simulation mode, either 'single' or 'dist' [single]
-o	Specify object file subdirectory [object/]
-p	Specify plan file subdirectory [plan/]
-t	Specify template location [($SODLHOME)/template/]
-v	Turns verbose mode ON [OFF]

Table 5-5 Command line options for sp

All subdirectories except *tmpltDir* and *cfgDir* are relative to *baseDir*. *RootProcess* is expected to be a process construct file in the *planSubdir* directory (relative to the *baseDir* value).

For user projects, the default directory structure is illustrated in Figure 5-2.

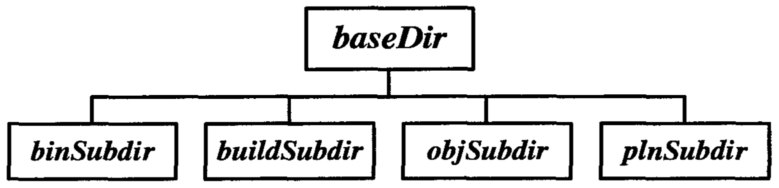


Figure 5-2 User project default directory structure

5.5. Configuration Files

The SODL parser uses a configuration file to specify various parameters for building the final product. These configuration files will differ from each other based upon the platform and compiler in use. The location of the configuration file defaults to $(SODLHOME)/config. Users can specify their own configuration file location using the -c option in the parser's command line invocation. The actual file name that the parser will look for in that directory is *mode.display.platform* where the –m option specifies the *mode* value, -d specifies the *display* component, and -l specifies the *platform*. Some examples are shown in Table 5-6.

Command Line	*Configuration file used*
sp -lcygwin ...	$(SODLHOME)/config/single.text.cygwin
sp -ctemp -llinux -dglut ...	./temp/single.glut.linux
sp -mdist –lcygwin -dtext ...	$(SODLHOME)/foo/single.text.cygwin

Table 5-6 Configuration file specification in the sp command line

This mechanism enables end user to specify their own graphics library and user interface, should they (perhaps wisely) opt out of the rather limited one provided with the SODL system. It also provides the means by which an end user can write their own simulation engine with which the code sp produces would interface.

The configuration files have key/value pairings that the parser and code generator can use to produce proper Makefiles. Here is a list of the keys and a description of how their values are used. These pairings take on the form *key-name* = "*key-value*".

Table 5-7 describes the key/value pair settings available to users to specify how the eventual product is built. Figure 5-3 has examples of some configuration files distributed with the SODL system.

When sp is run, it produces a number of files. There are several C++ source code files written to the directory specified –o option in the sp command line. There is also a Makefile written to *baseDir* used to actually build an executable simulation. A typical build will have one call to sp to produce the C++ files and the Makefile, followed by a 'make' to produce the final executable.

Key Name	***Value Meaning***
BIN	Location of the executable (relative to *baseDir*) produced as a result of complete SODL build process.(same as the –b option in the sp command line)
BUILD	Directory (relative to *baseDir*) to write the C++ files the SODL parser produces (same as -i in sp command line).
CC	Compiler command line invocation to use to compile C++ files.
CCFLAGS	Command line flags for the compiler. It is required to have as its last argument the option for naming the output file.
DISPLAY	Display type (same as -d option in SODL parser command line)
EXEEXT	Specifies the file extension for the executable image (required for Win32 to be .exe)
LD	Linker command line invocation to use to link the object files.
LDFLAGS	Linker command line options and flags. It is required to have as its last option the flag for naming the output file.
MODE	Simulation mode (same as the –m option in the SODL parser command line)
OBJECT	Directory in which the compiler places the object files after compilation (same as -o in SODL parser command line).
OBJEXT	File extension for the object files. It is normally ".o"
PLAN	Directory (relative to *baseDir*) where the root process declaration is located (same as -p in SODL parser command line).
REMOVE	Command for removing files (*rm* for Unix shells, *del* for MS-DOS like command shells)
TEMPLATE	Directory (relative to *baseDir*) where the SODL simulation engine and support source files are located (same as -t in SODL parser command line).

Table 5-7 Configuration file key/value descriptions

```
OBJEXT = ".o"
EXEEXT = ""
CC = "g++"
LD = "g++"
CCFLAGS = " -DLINUX -ftemplate-depth-64 -c -I/usr/include -O2 -o "
LDFLAGS = "-L/usr/X11R6/lib -lglut -lMesaGLU -lMesaGL -lXext -lX11 -lm -lXi -lXmu -o "
REMOVE = "rm -f "
```

a – $(SODLHOME)/config/dist.GLUT.linux

```
OBJEXT = ".o"
EXEEXT = ""
CC = "g++"
LD = "g++"
CCFLAGS = "-DLINUX -ftemplate-depth-64 -O2 -c -o "
LDFLAGS = "-o "
REMOVE = "rm -f "
```

b – $(SODLHOME)/config/dist.Text.linux

```
OBJEXT = ".o"
EXEEXT = ""
CC = "g++"
LD = "g++"
CCFLAGS = "-DLINUX -ftemplate-depth-64 -I/usr/include -O2 -c -o "
LDFLAGS = "-L/usr/X11R6/lib -lglut -lMesaGLU -lMesaGL -lXext -lX11 -lm -lXi -lXmu -o "
REMOVE = "rm -f "
```

c – $(SODLHOME)/config/single.GLUT.linux

```
OBJEXT = ".o"
EXEEXT = ""
CC = "g++"
LD = "g++"
CCFLAGS = "-DLINUX -ftemplate-depth-64 -O2 -c -o "
LDFLAGS = "-o "
REMOVE = "rm -f "
```

d – $(SODLHOME)/config/single.Text.linux

Figure 5-3 SODL configuration files for Linux platform

Chapter 6. SODL Language Structure

6.1. Overview

The Simulation Object Description Language (SODL) is designed to provide an enhanced event driven response representation for controlling entity activity in distributed discrete event simulations. It is a purely event driven language, and has some features of more traditional object oriented languages such as inheritance. SODL object descriptions are represented as a collection of stimulus/response handlers. That is, upon receipt of some stimulus, a simulation object will produce a possibly empty collection of internal and external responses to that stimulus. Here, stimuli are incoming messages, internal responses are state changes, and external responses are outgoing messages, as depicted in figure 6-1.

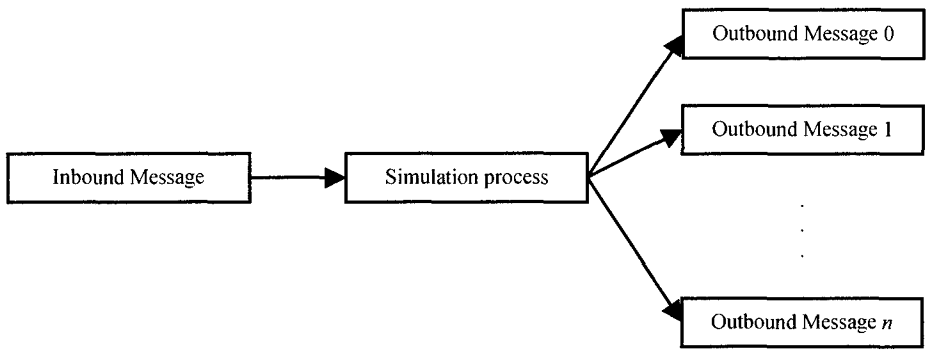


Figure 6-1 Depiction of the stimulus/response notion of a SODL process

This is not the case in SODL. The only mechanism provided for simulation object interaction is message passing between different instances. Methods are provided for individual instances to manipulate their own internal state.

6.2. Approach

SODL is a completely event driven language. It is heavily based upon C++, and relies upon many of the constructs of that language. Source code in SODL is passed through a parser and generates a collection of C++ files. These files are then compiled using a standard C++ compiler. Though the overall structure of the programming language is different from C++, the internal code executed when handling events is entirely C++. The build process is depicted in Figure 6-2.

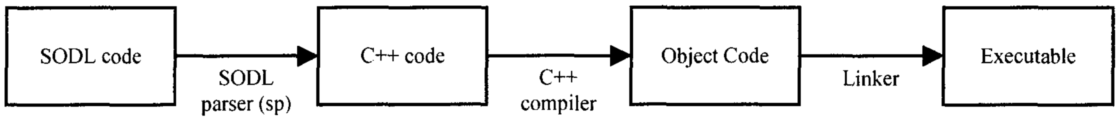


Figure 6-2 SODL project build steps

This approach is similar to others that have been employed for distributed simulation systems. In particular, both YADDES (Priess 1990) and APOSTLE (Wonnacott 1996) use this approach of translating a simulation specification from their respective languages, translating these user files into C and C++ files respectively, and then using standard compiling and linking tools to create an executable. It has the benefit of allowing systems to be ported to other platforms without having to write platform dependent binary code.

6.3. Constructs

A construct is the basic building block of SODL. In many aspects, a construct is roughly analogous to a C++ or Java class. Figure 6-3 shows the basic form of a construct.

```
{ construct-type:construct-name [ (parent-construct) ]
          ; |
          { construct-definition }
}
```

Figure 6-3 Basic construct form

Constructs are defined in files with a specific extension. Valid construct types and their associated file types are listed in Table 6-1.

Construct type	***File Extension***	***Description***
message	.msg	Defines a message that can be passed between object instances. It can contain data and method descriptions. Messages have an associated delivery time.
process	.proc	Defines behavior of a simulation object instance. It can receive and send messages, it has state variables, as well as method descriptions. It also contains stimulus/response definitions for handling messages.

Table 6-1 SODL Basic construct types

Optionally, constructs can inherit some functionality from a parent construct. Inheritance takes its form by enclosing the parent construct type into a set of parentheses following the declaration. Some constructs can optionally contain no functionality of its own, or extensive definitions of internal data, methods, and additional construct dependent definitions.

6.3.1. Message constructs

SODL files with .msg extensions contain message constructs. These message constructs are the means by which various processes within the simulation communicate with each other. They can have internal data, called a payload, and methods that act upon that payload. Sp creates two C++ files for each message it processes. The first is a header file, and the other is a source code file. Details on these files are described in Chapter 7. Entries for compiling them are placed into a Makefile for building the final executable. They are explored in depth in section 6.7. Sample message constructs are depicted in Figure 6-4.

```
{message:generic;}
```

a – generic.msg, a simple message with no data or methods.

```
{message:child_message(parent_message);}
```

b – child_message.msg, a simple message with no data or methods, with inheritance

Figure 6-4 Sample message constructs

6.3.2. Process constructs

```
{process:simple;}
```

a – simple.proc, a simple process with no data, methods or modes.

```
{process:child_process(parent_process);}
```

b – child_process.msg, a simple process with no data, methods or modes, inherited from parent.

Figure 6-5 Sample process constructs

SODL files with .proc extensions contain process constructs. A process construct has internal data, called a state, and methods that act upon that state data. Unlike message constructs, process constructs also have modes, which in turn have nodes. Each mode can be activated and deactivated independently. Each node can receive a message of fixed type, which changes the internal state of the process, and transmit messages. Nodes can only receive messages when their parent mode is active.

From one process construct, **sp** creates a pair of files to define in C++ the functionality of the process within the simulation engine. Details on these files are described in Chapter 7. Sample process constructs are depicted in Figure 6-5.

6.4. Import declarations

In order to make use of more than one construct, the programmer must reference them from within the body of all referencing files. This is done through an import directive. Import directives take on the form depicted in Figure 6-6.

```
{ import [ construct-type ]
        {
                construct-name1 [,construct-name2 [, ... ] ]
        }
}
```

Figure 6-6 Import directive specification

Import directives should be located at the top of a source code file, prior to the file's construct definition. There are two varieties of imports, the first imports SODL constructs for use within the importing construct, the other imports non-SODL declarations, such as C++ header files.

6.4.1. Importing SODL constructs

```
{import message {root_message} }

{message:start_sim(root_message);}
```

a – start_sim.msg; Imports message:*root_message* it to be used in the messages body.

```
{import message {start_sim, SetView} }
{import process {View3D} }

{process:root_process(View3D);}
```

b – root_process.proc; Imports message:*start_sim* and message:*SetView*, as well as process:*View3D*. process:*root_process* inherits functionality from process:*View3D*.

Figure 6-7 Sample import directives for message and process constructs

For importing SODL constructs, the *construct-type* in the import directive must match the *construct-type* in each of the files associated with the construct name. That is, when importing messages, the keyword message is used as the *construct-type*, and similarly for importing process constructs. This is necessary because different construct types are used differently in SODL, and the distinction in important. Figure 6-7 illustrates how some simple constructs import other constructs.

Message and process imports must be in either the *planDir* or the *tmpltDir*, as defined by the sp command line parameters.

It's not normally useful for message constructs to import process constructs. However, the language does provide for this possibility, even though any attempt to declare a process within a message produces an error when sp is run.

6.4.2. Importing non-SODL files

```
{import {<stdlib.h>, <stdio.h>} }
{message:start_sim(root_message);}
```

a – start_sim.msg; Imports stdlib.h and stdio.h for use in the start_sim message.

```
{import std {<vector>} }
{process:root_process;}
```

b – root_process.proc; imports the declaration for the *std::vector* class.

```
{import gvm {Node} }
{process:View3D;}
```

c – View3D.proc; imports the declaration for the *gvm::Node* class.

Figure 6-8 Sample non-SODL construct imports

Import directives also allow importing C++ header files with a way to allow compilation with external C++ source code. Figure 6-8 illustrates some examples.

Here Figure 6-8a imports the stdlib.h and stdio.h header files into the header file produced for the **message:*start_sim*** construct. This form (omitting the *construct-type*) should be used for any includes which are in the C++ global namespace.

Figure 6-8b imports the header file for the ***std*::*vector*<T>** class. In this case, we use this form for including headers for declarations in the ***std*** namespace.

Finally, figure 6-8c imports the header file gvm/Node.h, which must reside either in *planSubdir*/gvm or *in tmpltDir*/gvm. It will also compile into the final executable the file gvm/Node.cxx. Here any declarations should be in the ***gvm*** namespace.

6.5. Member Variable Declarations

Message and process constructs both allow declaration of member variables. Member variables are declared inside the construct declaration, and take on the illustrated in figure 6-9.

```
[namespace::]data-type:variable-name [ [] | [size] ] [ : initial-value ] ;
```

Figure 6-9 SODL construct member variable declaration

From this we can declare variables of variety of types, including arrays, each of which can be initialized with a certain value.

6.5.1. Basic data types

Data Type	***Description***
bool	Boolean value (takes one of the values {**true, false**})
byte	8-bit unsigned character value. Defined by **typedef unsigned char byte;**
char	8-bit signed character value
double	Double precision floating point value
float	Single precision floating point value
int	Single precision signed integer value
long	Double precision signed integer value
mtype	A Message type value. Defined by **typedef *sodl*::*Defs*::*MessageType sodl*::mtype;**
rand	A random number stream. Defined by **typedef *sodl*::*Random sodl*::rand**.
process	A handle to a process. Defined by **typedef *sodl*::*ProcessHandle sodl*::process;**
profile	A profiling tool class. Defined by **typedef *sodl*::*ProfileTools sodl*::profile**.
ptype	A Process type value. Defined by **typedef *sodl*::*Defs*::*ProcessType sodl*::ptype;**
uint	Single precision unsigned integer value. Defined by **typedef unsigned int uint;**
ulong	Double precision unsigned integer value. Defined by **typedef unsigned long ulong;**

Table 6-2 SODL construct member variable basic types

Member variables can take on any of the basic types listed in table 6-2. Each of these data types has exactly the same properties as the C++ data types of the same name, or typedef as the case may be. In fact, they are instantiated as those very same C++ data types.

Table 6-3 provides some sample declarations and their meaning within the SODL language.

SODL Declaration	*Description*
int:x;	Creates an integer value named **x** with an undefined initial value.
double:y(0.0);	Creates a double precision value named **y** initialized to 0.0
float:z[3](1.0);	Creates an array of 3 floating-point values, each initialized to 1.0.
char:x[];	Creates an empty array of characters. Initializer cannot be used in this case.
long:y[4];	Creates an array of 4 uninitialized long integers with undefined initial values.

Table 6-3 Sample SODL member variable declarations

```
{
        message:start
        {                       // message:start
                int:x;          // Single precision integer
                double:y(0.0);  // Double precision floating-point number initialized to 0.0
                float:z[3](1.0); // Array of 3 floats each initialized to 1.0
        }                       // message:start
}
```

a – member variables constituting the payload of message:start instances

```
{
        process:root
        {                       // process:root
                char:x[];       // Uninitialized array of characters of unspecified length.
                long:y[4];      // Uninitialized array of 4 long integers
        }                       // process:root
}
```

b – member variables constituting the state of process:root instances

Figure 6-10 Sample member variable declarations in SODL constructs

Arrays are implemented using the C++ Standard Template Library's ***std::vector*<T>** making them somewhat more flexible than the traditional C-style technique of using a pointer. Pointers can still be declared if programmers explicitly state the namespace (i.e. the global namespace). This technique is covered in section 6.5.2. Some sample variable declarations are provided in figure 6-10.

6.5.2. Extended data types

You can alternately create structures or other data types in a C++ namespace (including the global namespace). By explicitly specifying the namespace, programmers can provide instances of any variable type that could be instantiated in a C++ program. Any structures in namespaces (including the global namespace) need to be retrieved through an import directive. Some sample variable declarations are shown in Table 6-4.

Declaration	***Import Needed***	***Description***
std::set<double>:x;	{import std {<set>} }	***std::set*<double>** instance named **x**, uninitialized.
std::string:y("Hello")	{import std {<string>} }	***std::string*** instance named y, initialized to "Hello" .
GLenum:mode[4](GL_POLYGON);	{import {<GL/glut.h>} }	An array of 4 ***GLenum*** 's named mode, initialized to ***GL_POLYGON***.
gvm::Node*:gr_node(NULL);	{import gvm {Node} }	An instance of ***gvm::Node**** named ***gr_node***, initialized to ***NULL***.

Table 6-4 Sample extended data type declaration

As in the standard data types, arrays are instantiated as ***std::vector*<T>**.

6.5.3. Process constructs as data members

It is also possible to declare processes as variables within process construct declarations. These declarations can take on either of the two forms shown in Figure 6-10.

When a process is declared within a process construct, it is important to note that the variable associated with the process declaration is only a handle to the actual process instance, and not the instance itself. No methods or internal data can be accessed through this handle. The handle acts as an address for message delivery, and for filtering incoming messages. It does not have any type information associated with it, though a typed handle can be resolved to gather that information.

When the first form in figure 6-11 is used to declare a process, an actual instance (or collection of them in the case of arrays) of the type specified by the construct name field is created. The simulation engine specified in the *node-distribution* field will then manage the activities of its instances. This form can only be used in process construct declarations, and not within message constructs.

construct-name:*variable-name* [**[** *size* **]**] [: *node-distribution*] ;
process:*variable-name* [**[]** | **[** *size* **]**] ;

Figure 6-11 Process member variable declaration

The second form allows the variable to act as a placeholder and does not actually create any process instances. Instead, an empty handle (or an array of them) is created which allows local storage of references to arbitrary process instances. This form can be used in either process or message constructs. Some sample process declarations are listed in Table 6-5.

Declaration	***Import Needed***	***Description***
***ball*:b[1000];**	{import process {ball} }	Array of 1000 handles to **process:*ball*** instances named **b**.
***Node3D*:*n*;**	{import process {Node3D} }	Handle to a **process:*Node3D*** named **n**.
process:*shape*[];		Uninstantiated and unspecified process handle array.

Table 6-5 Process construct declarations

Process instances are statically assigned to a specific simulation engine for the duration of the simulation run[9]. The *node-distribution* field can be used to specify which simulation engine controls each of the instances declared in the construct. When the node-distribution is omitted for a process declaration, the construct will be instantiated and controlled in the engine where the construct declaring the process is controlled.

```
engine-node-number
< element-node-equation >
```

Figure 6-12 Forms for specifying controller simulation engine

The two forms for specifying the node-distribution are shown in Figure 6-12. The first form can be used either with single instances or on arrays or processes. It specifies that the process instance (or all of them in the case of an array) be controlled by the *engine-node-number.*

The second form allows for different elements of an array of processes to be controlled by different simulation engines. This second form allows general C++ code which, when evaluated, produces an unsigned integer value, to be placed between angle brackets. There is also a macro substitution for the characters '@' and '#'. The '@' character evaluates to the index in the array of the process. The '#' evaluates to the total number of elements in the array. Each element is then assigned to the engine to which its *element-node-equation* evaluates. Some examples are listed in Table 6-6.

[9] There is no process migration in the current SODL simulation engine, though there is nothing to preclude this feature from future implementations.

Process declaration	***Description***
View3D:view:3;	A new **process:*View3D*** instance is created on simulation engine 3.
Node:nodes[200]:10;	An array of 200 **process:*Node*** instances is created on simulation engine 10.
Simple:simple[10000]: @%((long) sqrt(#));	An array of 10000 **process:*Simple*** instances. Engine i%100 controls **simple[i]**.

Table 6-6 Engine specification for process declaration

One final note here is that message constructs cannot have as data members explicitly typed process declarations. That is, only the second form in figure 6-10 is allowed within a message construct.

6.5.4. A note on references and pointers

Because of certain constraints associated with the Time Warp algorithm, C++ references, although ignored during the initial processing of SODL files (i.e. with sp), may cause problems with C++ compilers. This stems from the fact that copying class instances containing as member variables references causes some (perhaps all) compilers to complain without explicit declaration of the copy constructor. The decision to forgo this declaration (for performance concerns) has made it necessary to likewise forgo the use of references.

Pointers may be used in lieu of references, though they are stylistically and functionally inferior to references in C++. Still, even though permitted, their use should generally be eschewed. That's not to say that there is no use for them; a pointer was used for the GLUT view manager as a means of allowing a process to reference the same view instance regardless of its timestamp. This provided a significant performance improvement while retaining a suitably generalized mechanism for producing graphical output.

Stepping back to consider the implications of a pointer in a general SODL environment, one sees that any uses of a pointer must be atemporal in their nature when defined in terms of a process construct, and highly questionable in terms of a message construct. In the case of a message construct, a pointer really has no meaning since the destination of the message may be a process instance located on a different host machine. In the case of a process construct, only the pointer is copied to each process instance during the state saving portion of the Time Warp algorithm; the data that the pointer points to is not copied.

Therefore, unless arrangements are made to reallocate data at each of these temporal transitions, all process instances associated with a particular process (i.e. the same process at different points in time) point to the same memory location and can interact with the data in that memory location atemporally.

6.6. Method Declarations

Both types of constructs can also declare methods. For messages, these methods behave the way they do in most object oriented programming languages. That is, a programmer can call a method defined within a message construct declaration. However, in process construct, since both messages and processes only have handles to other processes, there is no way to call the methods of another process. The reason for this is that a process may be instantiated on a remote simulation node, and not directly accessible to the local process instance. Methods may be declared as indicated in figure 6-13.

```
method:method-name(access-specifier; return-type; [ variable-specifier; [variable-specifier; [ ... ] ] ] )
{
        method-body
}
```

Figure 6-13 General method form

The *access-specifier* is one of {**public**, **protected**, **private**}. It tells the parser what sort of access to the method is allowed. The meanings are analogous to the respective C++ keywords.

The *return-type* is simply a C++ data type or any of the data types listed in Table 6-2. Any types not listed in Table 6-2 need to be prefaced with their C++ namespace identifier, including any types defined in the global namespace (in this case they need to be prefaced with '::')

Method parameters are defined just like member variables, as described in Figure 6-9. They also require that any non-standard data types be prefaced with the identifier for the namespace in which they were defined.

The method-body is nothing more than some C++ code that performs the desired function of the method. This code is cut from the SODL program and pasted directly into the resulting C++ source code files

Method names must be unique within the process or message in which they are declared.

```
{
        message:start
        {                                                   // message:start
                double:x[10];                               // Uninitialized array of 10 doubles

                method:getX(public; double; uint:i;)
                {                                           // method:getX(public; double; uint:i;)
                        return (i<x.size()) ? x[i] : 0.0;   // Return x[i] if i is in [0, x.size())
                }                                           // method:getX(public; double; uint:i;)

                method:setX(public; void; uint:i; double:v;)
                {                               // method:setX(public; void; uint:i; double:v;)
                        if (i<x.size()) x[i]=v; // Set x[i] to v if I is n the proper range
                }                               // method:setX(public; void; uint:i; double:v;)

                method:init(public; void;)
                {                                                     // method:init(public; void;)
                        for (int i=0; i<x.size(); ++i) x[i]=0.0;      // Initialize all elements of x to 0.0
                }                                                     // method:init(public; void;)
        }                                                   // message:start
}
```

Figure 6-14 Sample methods

A sample method declaration for a message construct is presented in figure 6-14. Process constructs handle methods in exactly the same way.

6.7. Messages

```
{ message:message-name[ (parent-message) ]
        ; |
        { message-definition}
}
```

Figure 6-15 Message construct form

Messages are packets of information passed between process instances and provide the only mechanism for inter-process communication. Each message has a source process and a collection of destination processes. They flow from the source process to the processes in the destination list. Figure 6-15 shows the general form of a message construct. Figure 6-16 shows how the messages flow from source to destination in detail.

The *message-name* is a unique type identifier for the message construct being declared. All message instances have as their type one of the ***Defs::MessageType*** enumerators, ***Defs::SMT_message-type***. The *parent-message* allows for inheriting the data members and methods of the *parent-message* message type.

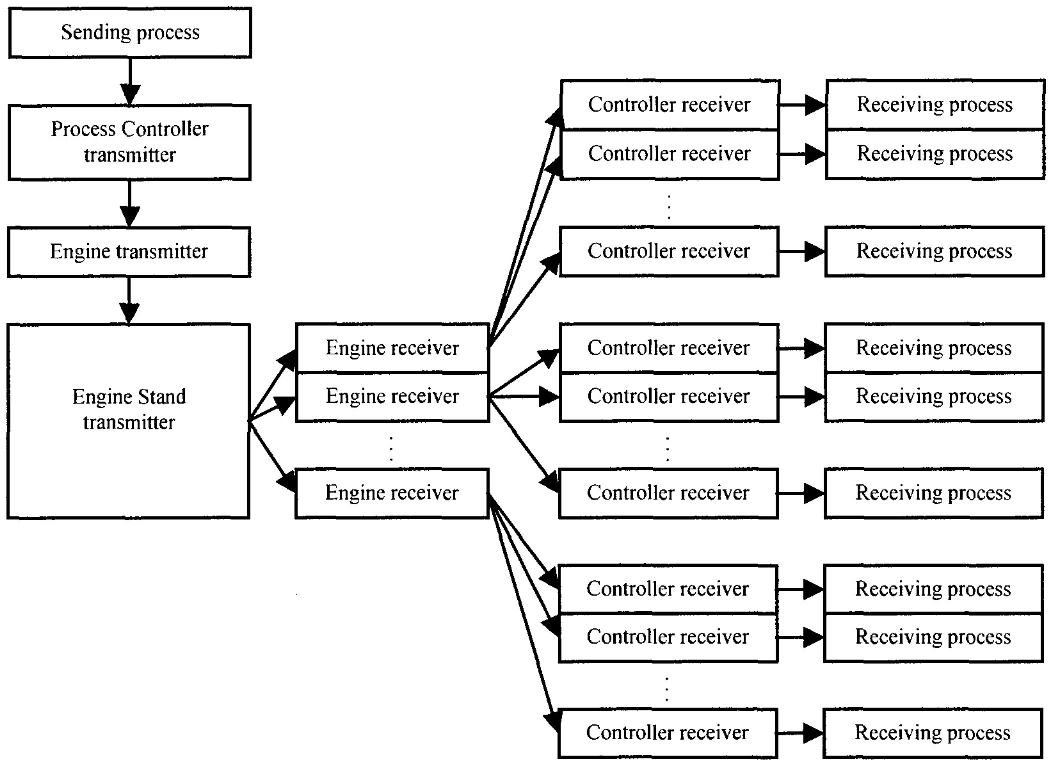


Figure 6-16 Message flow from sending process to receiving processes

Messages contain a collection of system-defined parameters, and possibly a user-defined payload and collection of methods. The *message-definition* is the user-defined portion of the message definition and consists of these member variables and methods declared as indicated in sections 6.5 and 6.6 respectively.

In addition to the user-defined portion of a message, there are a number of system-defined methods and member variables.

6.7.1. System-defined message member variables

The system-defined member variables allow users to customize certain aspects of message delivery. Though these variables should be manipulated through the provided accessor functions described in section 6.7.2.

The member variables in Table 6-7 are declared as protected (except for *genTime*, which is private) and non-static.

Message member variable	***Description***
destination_list:dest	This is the list of destination processes for the message. ***destination_list*** is a typedef of ***std*::*map*<ulong, *std*::*set*<ulong> >**.
double:*genTime*	Simulation time at which this message was generated.
message:*me*	A unique identifier for the owning message instance.
ulong:*node*	Engine node number controlling the message instance. This may be different from the engine node number where the message was actually generated.
bool:*preempt*(false)	The user can, at compile time specify a collection of destinations for the message. If *preempt* is set to **true**, the message is not sent to those destinations. Any destinations added during the handling of the message are retained.
process:*source*	This is shorthand for the source process of this message. *source.first* is the node in the distributed simulation managing the source process. *source.second* is the specific index of the source process.
double:*time*	Simulation timestamp for message delivery.
bool:*tx*(true)	The user can set *tx* to **false** to prevent message transmission. *tx* defaults to **true**.

Table 6-7 System-defined message member variables

6.7.2. System-defined message methods

Access to the member variable listed in Table 6-7 should be performed through the member functions, rather than direct manipulation of them. Table 6-8 provides a list of useful system-defined message methods, some of which can be overloaded to change their behavior.

Each of the methods can be overloaded in a message construct declaration, though the only two methods where this serves any clear purpose are *init* and *getTX*.

In the case of *init*, the default behavior does not do anything; it is an empty function. However, some initialization of member data may not easily be initialized at the location of its declaration. The init method is provided as a means to address that shortcoming.

The *getTX* function can be overloaded to provide a means to test certain conditions that will either permit or refuse transmission of the message.

In all cases of method overloading, users should ascertain whether they desire the default behavior of the parent construct to be an aspect of the child construct's behavior. If so, calls to these parent construct versions need to be explicitly made from within the construct method of the child construct.

Message method	***Description***
method:*addDest*(public; void; process:*p*;)	Adds *p* to the list of destinations.
method:*addDest*(public; void; *std*::*vector*<process>:*p*;)	Adds *p*[*i*] to the list of destinations, *i*<*p*.*size*()
method:*clearDest*(public; void;)	Clears the message's destination list, and sets *preempt* to **true**.
method:*getGenTime*(public; double;)	Returns *genTime* to the calling routine.
method:*getID*(public; *sodl*:: *Handle*;)	Returns a copy of *me* to the calling routine.
method:*getNode*(public; long;)	Returns *node* to the calling routine.
method:*getSource*(public; process;)	Returns *source* to the calling routine.
method:*getTime*(public; double;)	Returns *time* to the calling routine.
method:*getTX*(public; bool;)	Returns **true** if and only if *tx* is **true** and the destination list is not empty. It can be overloaded to check for additional criteria for message transmission.
method:*getType*(public; mtype;)	Returns the message type to the calling routine.
method:*getTypeName*(public; *std*::*string*;)	Returns a string representation of the message type to the calling routine.
method:*init*(public; void;)	An initializer that is called immediately after message generation. It can be used to perform specific initialization.
method:*isPreempted*(public; bool;)	Returns *preempt* to the calling routine
method:*isType*(public; bool; mtype:*t*;)	Returns **true** if and only if the message instance is of type (or a subtype of) *t*.
method:*setPreempted*(public; void; bool:*p*;)	Sets the *preempt* flag to *p*.
method:*setTime*(public; void; double:*t*;)	Sets *time* to *t*. This is adjusted during message transmission to ensure that it occurs at some time after the message generation time (*genTime*).
method:*setTX*(public; void; bool:*v*;)	Sets the *tx* flag to *v*.

Table 6-8 Common system-defined message methods

6.7.3. System-defined messages

Message name	***Description***
message:*Antimessage*	***Antimessage*** instances are paired with message instances and saved on the simulation engine that owns the process that is the source of the message. An ***Antimessage*** is transmitted when a rollback causes the engine to revoke messages that have previously been transmitted. When a message and its associated ***Antimessage*** are combined, they annihilate each other, ensuring that the original message is never delivered.
message:*EndSimulation*	***EndSimulation*** messages are time stamped with the end simulation time (1e307) and are intended to be delivered to all simulation processes by the time the simulation is complete. This has the affect of setting the system clock of each engine on an engine stand to the end simulation time when it has no messages remaining to process
message:*StartSimulation*	Each process in the simulation receives a ***StartSimulation*** message when the simulation begins. This is time stamped with some time prior to 0 (defaults to –1). This allows individual processes to perform some last minute initialization and set up prior to actually beginning the simulation run and as a bootstrapping device.

Table 6-9 System-defined messages

A number of messages are used within the run time system to perform various functions. Users defined processes can, in some instances receive these messages. Each is defined in detail in the following sections. Additional messages are defined for the GLUT view manager, and are discussed in more detail in Chapter 9. These system-defined messages are listed in Table 6-9.

6.7.4. Message Handles

Each message has a message handle, ***me***, containing identifier information. This is a ***sodl::MessageHandle*** instance (which has a typedef to **message**) and contains the index of the engine where the associated message was generated, and the message instance count for that message. Each message generated has a unique message handle. Users should not modify these values, as it may cause problems with message revocation.

6.8. Processes

Process can be thought of as a self-contained package of state information that responds to incoming messages. The process will change its state information and send new messages in response to an incoming message. Messages are processed in time stamp order. The process takes on the time stamp value of the last message it processed.

```
{ process:process-name [ (parent-process)]
        ; |
        {
                process-definition
        }
}
```

Figure 6-17 Process declaration syntax

Processes are isolated message handlers. They are unable to communicate with each other directly, and must rely exclusively upon message passing to perform this function. Process declarations have the form specified in Figure 6-17. The *process-name* is a unique type identifier for the process class being declared. All process instances have as their type one of the ***Defs::ProcessType*** enumerators, ***Defs::SMT_process-type***. The *parent-process* allows for inheriting the data members and methods of the *parent-process* process type. The *process-definition* is the user-defined portion of the process definition. It consists of member variable and method declarations as described in sections 6.5 and 6.6. It is in this process-definition section that we also define the process modes and node, which allow the process to handle messages.

The C++ code sp generates declares one class for each process type defined named ***sodl*::*process-name***. It performs the functions associated with the process. Instances of these classes receive messages and manage the responses to those messages. The structure and format of these files is discussed in more detail in Chapter 7.

6.8.1. System-defined process member variables

There are a number of system-defined member variables associated with a process. These member variables, listed in Table 6-10, may be useful in governing a process response to incoming messages. Other system-defined variables are accessed through the accessor functions in section 6.8.3.

Process Variables	***Purpose***
process:*me*	An identifier for this instance of the process. Useful for addressing messages to owning process. The *resolve* method can be called to get a reference to the actual handle associated with the process.
rand:*random*	A random number stream. This is a static member in the ***sodl*::*Process*** declaration, so all such instances share the same random number generator (RNG). Programmers can provide their own RNG instance (as a process member variable) in the process if desired.
mtype:*type*	Type information regarding the specific process instance.

Table 6-10 System-defined process member variables

6.8.2. System-defined process methods

There are several system-defined methods useful in ascertaining or establishing certain parameters related to a process state. The methods listed in Table 6-11 are intended to assist the programmer in making decisions regarding the process state and to perform various tasks associated with process management. Most of the methods are not intended to be overloaded, and should not be without great care taken. The exceptions to this are methods ***backup***, ***fossilCollect***, ***init***, and ***restore***. These methods provide mechanisms for dealing with initialization (***init***) and as a means by which a process can interact with the underlying simulation synchronization protocols.

Process System Define Methods	***Purpose***
method:*backup*(public; void;)	When the process is backed up for the purpose of sate saving, the time stamp of the new state is set and the backup method is called. If programmers want to perform some action at this point, they may overload the backup methods to perform it.
method:*fossilCollect*(public; void;)	When a process state is about to be fossil collected, its fossilCollect method is first called. Programmers may overload this method to perform any required output or irrevocable action prior to actual fossil collection.
method:*getEngine*(public; *sodl::Engine&*;)	Returns a reference to the engine controlling this process instance to the calling routine.
method:*getIndex*(public; long;)	Returns to the calling routine the unique index on the simulation engine for the associated process instance.
method:*getNode*(public; long;)	Returns to the calling routine the simulation engine controlling the associated process instance.
method:*getType*(public; ptype;)	Returns to the calling routine the specific type of the associated process instance.
method:*getTime*(public; double;)	Returns the process timestamp of this instance to the calling routine.
method:*init*(public; void;)	Overloading init allows initialization. It is called shortly after process instantiation, and before the first simulation message is delivered.
method:*isType*(public; bool; ptype:*t*;)	Returns to the calling routine true if and only if **this* is an instance of a process of type *t*. This includes subclasses of type *t*.
method:*restore*(public; void;)	When a process state is restored due to a rollback, the process controller calls the restore method to allow the process the opportunity to perform any functions that might be required.

Table 6-11 System-defined methods for process classes

6.8.3. Process Handles

Method	***Description***
ptype *ProcessHandle::getType*(void)	Returns the type for the ***sodl::Process*** instance associated with this ***sodl::ProcessHandle*** instance.
bool *ProcessHandle::isType*(ptype *t*)	Returns true exactly when the ***sodl::Process*** instance associated with this ***sodl::ProcessHandle*** is of type *t*.

Table 6-12 sodl::ProcessHandle type routines

Process handles are unique identifiers for process instances. Each process instance has its handle in the member variable, ***me***. This is of type sodl::ProcessHandle, and has an associated typedef, process. Each handle contains engine and instance index information used to identify a specific process instance. Process handles can also be used as a reference to process instances. In this case, type information can be derived from the process handle instance. Methods to perform this sort of type-query are listed in Table 6-12.

6.8.4. Special Processes

There are no special system-defined processes, in the sense. However, the *root process* occupies a unique position in the simulation system. It is always instantiated and controlled on engine 0 in the distributed simulation, and has index 0. It is the base process specified in the sp command line argument.

6.9. Mode and Node Declarations

Only process constructs may define modes, and nodes within those modes. A mode may be either active or inactive. Only nodes defined within active modes can process messages. This feature is useful if there are a different behaviors associated with different process modes. For instance, suppose a user wanted to look at the dynamics associated with an ant colony. Individual ants might have different functions that change with respect to certain external stimuli. At one point, the ant might be out looking for food. At another time, it might be taking the food back to the nest. Still another time it might be defending the nest from an intruder. Each of these different behaviors can be activated and deactivated inside the code simulating the ant's behavior, and messages will only be delivered to nodes that reside in active modes.

Each node is declared so that it responds to a specific message type. These nodes produce state changes within the receiving process and subsequently transmit a (possibly empty) collection of output messages in response.

6.9.1. Modes

Modes provide a mechanism for easily changing process behavior without the programmer having to explicitly perform a number of checks, and handle the message differently depending upon certain internal state data. Each mode in a process construct has a unique name corresponding to a process member variable of type ***sodl::ProcessMode***, a C++ class defined as part of the underlying SODL system. The format for declaring a mode is illustrated in Figure 6-18. ***sodl::ProcessMode*** instances have methods, listed in Table 6-13, for managing the mode state.

```
mode:mode-name
        {
                node-declarations
        }
```

Figure 6-18 Mode declaration syntax

Modes are inherited from parent processes, though the mechanism governing this inheritance may not be immediately obvious. Let process construct *a*, with a mode named *m*, be a sub-process (i.e. inherited from) of process construct *b*, also with a mode named *m*. In the class declarations sp generates, ***a***::***m*** and ***b***::***m*** are actually the same declaration. That is, only ***b***::***m*** is declared; any changes to m from a methods or node in a change the declaration in the parent class. Thus, whatever state changes are made from the perspective of *a* are also made from the perspective of *b*, and vice versa.

Mode Methods	***Purpose***
bool *sodl*::*ProcessMode*::*isActive*(void)	Returns **true** if the mode is active or **false** if it is not.
void *sodl*::*ProcessMode*::*setActive*(bool *a*)	Activates (*a*=**true**), or deactivates (*a*=**false**) the mode

Table 6-13 Mode system-define methods

When the ***setActive*** method for a mode is called, the change does not immediately occur. The change will be finalized only when the process clock is advanced. The reason for this is that there may be multiple messages with the same timestamp addressed for the same process. If one of them changes the active flag for a mode, and were that change to be immediately reflected, handling of the other message (if it's done afterwards) might be affected. By waiting until the process time stamp is advanced to commit changes to the active flag, all messages will be processed consistently vis-à-vis the mode active flags, regardless of the order in which the messages are actually handled.

6.9.2. Nodes

```
node:node-name [ message-type:input-message-name ]
               [ output-message_0, output-message_1, ..., output-message_n ]
               {
                       node-body
               }
```

Figure 6-19 Node declaration syntax

Nodes are defined within a mode. No two nodes within the same mode declaration may have the same name, though nodes in different modes may. The declaration format for a node is illustrated in Figure 6-19. The form for output message specifications is depicted in Figure 6-20.

```
output-message-type:output-message-name
        [
                [ [ size-specifiaction ] ]
        ]
        [
                => (destination0; destination1; ...; destinationd;)
        ]
        [
                :(time-specifier)
        ]
```

Figure 6-20 Output message form

6.9.2.1. Input message

```
{import message {Generic} }
{import std {<iostream>} }

{
        process:Simple
        {                                                       // process:Simple
                mode:Default
                {                                               // mode:Default
                        node:runner[Generic:in][]
                        {                                       // node:runner[Generic:in][]
                                std::cout << in.data << std::endl;  // Display input data
                        }                                       // node:runner[Generic:in][]
                }                                               // mode:Default
        }                                                       // process:Simple
}
```

Figure 6-21 Stand along input message usage

Nodes are defined to accept all messages of type *message-type* that are transmitted to the owning process instance. The node will also accept messages of types derived from *message-type*. So, if node *n* accepts messages of type *m*, and message *p* is a sub-construct of message *m*, any messages of type *p* will also be handled in node *n*. *p* will be first cast to type *m*, and any methods or member variables in type *p*, not in type *m* will not be accessible to *n*[10].

[10] In fact such attempts at a dynamic cast will probably lead to an abnormal program, termination since the messages are passed to the node by value, and not by reference.

From the programmer's perspective, a variable with name input-message-name acts as the interface to the message. It is a message instance that is passed to the node handling the message and has a parameter of type *input-message-type* with name *input-message-name*. An example of how declare and interact with an input message is depicted in Figure 6-21.

6.9.2.2. Output messages

The output message format allows programmers the flexibility of compactly specifying the default number of messages, default destinations, and default time stamps. In most cases these call all be overridden (the only exception being that the array form must be used in order to send multiple instances of the same message type from the output message specifier).

6.9.2.2.1. Default output message form

```
{import message {Generic} }
{import std {<iostream>} }

{
        process:Simple
        {                                                          // process:Simple
                mode:Default
                {                                                  // mode:Default
                        node:runner[Generic:in][Generic:out]
                        {                                          // node:runner[Generic:in][]
                                std::cout << in.data << std::endl; // Display input data
                                out.addDest(me);                   // Return the message to me
                                out.setTime(in.getTime()+1.0);     // Schedule for later return
                        }                                          // node:runner[Generic:in][]
                }                                                  // mode:Default
        }                                                          // process:Simple
}
```

Figure 6-22 Explicit specification of destination & timestamp in body

The default output message form does not specify a size, destination, or timestamp. In this case, the process creates exactly one message instance of *output-message-type*, named *output-message name*. The destination list is empty, meaning that it will not be delivered to any process unless some destinations are added via the *output-message-name.**addDest***(...) method. The default time stamp is also used, which is some point after the time stamp value in the process handling the message. The message timestamp can be set using the *output-message-name.**setTime***(...) method. It accepts a double precision floating point number serving as a timestamp for the message. Figure 6-22 modifies somewhat the code in Figure 6-21 to demonstrate how a node may customize the message internal data.

If the output message time stamp is not explicitly set anywhere in the body of the node, the simulation engine will set it to a value slightly larger than that of the input message. The actual value of this timestamp is given in Equation 6-1.

$$next-time(t) \equiv \begin{cases} t < 0, & (1-10^{-15}) \cdot current-time \\ t = 0, & 10^{-307} \\ t > 0, & (1+10^{-15}) \cdot current\_time \end{cases} \tag{6-1}$$

6.9.2.2.2. Output message size specification

```
...
        node:runner        [Generic:in]                     // Upon receipt of an input message
                           [Generic:out1[],                 // Create unspecified number of outputs
                            Generic:out2[5]]                // And an array of five of them
        {                                                   // node:runner[Generic:in][...]
               for (int i=0; i<out2.size(); ++i)            // Loop over the output arrays
               {
                           out1.push_back(me);              // Create a new message with source me
                           out1.back().addDest(me);         // Add me as a destination process
                           out1.back().setTime(getTime()+i); // Schedule for later return
                           out2[i].addDest(me);             // Add me as a the destination process
               }                                            // for (int i=0; i<out2.size(); ++i)
        }                                                   // node:runner[Generic:in][...]
...
```

Figure 6-23 Message count specification for output message arrays

The output message size specification allows the programmer to create multiple message instances each independently addressed, time stamped, and to have differing payloads. When either the '[]' or '[*size-specification*]' form is used, *output-message-name* is passed to the node as a ***std::vector<****output-message-type*> instance. In the first case, the vector is empty, and new messages can be added to the vector by calling *output-message-name**.push_front(me)***. Messages can also be removed after their creation if the for some reason do not merit additional consideration. This can be accomplished through use of the ***std::vector<*T*>.pop_front*()** method.

This can also be done with the '*[size-spcification]*' form if additional messages are required, or the programmer decides not to send some.

Figure 6-23 further modifies the code in the previous two figures (omitting the redundant code) to demonstrate how to use either form. In the case of the messages in the *out2* array, their timestamps will be given by the default value defined by Equation 6-1. This means that it will take a very long time before any of the *out1* messages will be processed. This is in fact going to lead to the situation where the memory of the computer is eventually consumed by pending messages leading to an abnormal program termination, quite likely prior to any of the *out1* messages ever being processed. The purpose of this code segment, and many that follow is to provide some insight into the mechanisms for controlling message delivery, not necessarily to provide a logical framework for nodes in systems that will eventually do anything useful.

6.9.2.2.3. Destination specification

Each output message can have an arbitrary number of default destinations. Each destination needs at run time to resolve to a **process** or ***std::vector*<process>** instance. For arrays of output messages, this might be somewhat limiting, so a mechanism for independently addressing different elements of an array of messages, similar to that used in the initialization of arrays of variables, has been provided to allow programmers this flexibility in a compact form. Any '@' characters will be replaced with the index of the array element, and any '#' will be replaced with the size of the message array.

```
...
        node:runner     [Generic:in]                          // Upon receipt of an input message
                        [Generic:out1[]=>(process(0,@);),     // Create unspecified number of outputs
                         Generic:out2=> (me;)]                // And an individual one
        {                                                     // node:runner[Generic:in][...]
                for (int i=0; i<5; ++i)                       // Loop over the output arrays
                {
                        out1.push_back(me);                   // Create a new message with source me
                        out1.back().setTime(getTime()+i);     // Schedule for later return
                }                                             // for (int i=0; i<5; ++i)
        }                                                     // node:runner[Generic:in][...]
...
```

Figure 6-24 Adding default destinations to output messages

Figure 6-24 shows an example of how this default addressing is specified. In the case of *out2*, (which has changed from an array to a single message instance) the destination is explicitly declared in the node header. In the case of out1, a C++ expression resolves to a process instance. In particular, message *out1*[*i*] is sent to **process**(0,*i*), which is the *i*th process on simulation engine 0 (which for the moment we will assume actually exists). In both cases, additional destinations can be added, each terminated with a semicolon.

The expressions serving as message destinations are actually evaluated and added to the destination list after the node has finished processing. The reason for this is that any changes to local variables used in computing the destinations might be modified in the node body. However, the programmer can preempt adding these destinations by calling the message's ***clearDest***() method. This will clear any destinations previously added to the destination list while setting a flag that is checked prior to adding the default destinations.

6.9.2.2.4. Time stamp specification

```
...
        node:runner        [Generic:in]                       // Upon receipt of an input message
                           [Generic:out1[]:(getTime()+@),     // Create unspecified number of outputs
                            Generic:out2:(getTime()*2.0)]     // And an individual one
        {                                                     // node:runner[Generic:in][...]
                 for (int i=0; i<5; ++i)                      // Loop over the output arrays
                 {
                           out1.push_back(me);                // Create a new message with source me
                           out1.back().addDest(me);           // Add me as a destination process
                 }                                            // for (int i=0; i<5; ++i)
                 out2.addDest(me);                            // Add me as a the destination process
        }                                                     // node:runner[Generic:in][...]
...
```

Figure 6-25 Default time stamp specification

As with destinations, output messages can also have a default time stamp. It can also be set differently for each message instance in an array. We again substitute the array element index for the '@' in the time stamp specification, and the array size for the '#' character. We see this illustrated in Figure 6-25.

Here we have each of the message time stamps for *out1*[*i*] set to ***getTime***()+*i*. For out2, we specify the message time stamp to ***getTime***()*2.0 (assuming that the current time is strictly positive).

As in the destination specification, the default time stamp value is set after the node complete processing the input message. If in the course of handling the input message, the node has code to alter the default time stamp value, an internal flag is set in the message indicating that it should not attempt to set the message time stamp with the default value.

6.9.2.2.5. Combining destination and time stamp specifications

The default time stamp and destination specifications can both appear in the output message declaration provided they appear in the order they appear in Figure 6-19. An example of this is shown in Figure 6-26.

```
...
        node:runner    [Generic:in]                           // Upon receipt of an input message
                       [Generic:out1[]=>(process(0,@););:(getTime()+@),
                       Generic:out2=>(me;):(getTime()*2.0)]
        {                                                     // node:runner[Generic:in][...]
                for (int i=0; i<5; ++i)                       // Loop over the output arrays
                       out1.push_back(me);                    // Create a new message with source me
        }                                                     // node:runner[Generic:in][...]
...
```

Figure 6-26 Combined default destination and time stamp specification

6.9.2.3. Node body

As in methods, the body of the node declaration is merely a block of C++ code. This code dictates how the node responds to the incoming message. This response may include creating state changes in the parent process and formatting output message payloads, time stamps, destinations or other aspects of message formatting.

Chapter 7. C++ code generation

When the SODL parser (sp) is run, it generates a number of files in *baseDir/bldSubdir*. For each message and process file, two C++ source code files are created, one with header information and one with the actual method definitions. Three additional files, *Defs.h*, *Defs.cxx*, and *Main.cxx* are also created.

Both process and message constructs in the SODL language become C++ classes in the resulting output files. Data members and methods within the SODL constructs are incorporated into the resulting C++ classes in the obvious fashion of direct inclusion. There is also a great deal of functionality that must be incorporated into the classes to ensure they properly interface with the simulation engine.

The SODL parser also creates entries into a Makefile in *baseDir* with a largely correct dependency specification so that minor changes to one file do not normally necessitate rebuilding the entire simulation system. Additionally, in a further effort to reduce unnecessary builds, C++ files are not written to the hard drive if doing so would not change the contents of the resulting file.

One other point that should be made here is that it is not immediately obvious how to create a universal library of the simulation engine that can be linked (either statically or dynamically) to create a final executable. The reason for this is that there is a great deal of type information generated during the SODL parser run needed within the SODL run-time system. Possible future enhancements include eliminating this restriction, due to the lengthy time required to rebuild the run-time system for each simulation project, or each time a given project changes in some substantive manner.

7.1. Message construct files

Member data and methods from message constructs are incorporated into the resulting C++ files straightforward manner. The message class declaration provides the necessary functionality to operate properly with the SODL run-time system.

7.1.1. A simple message construct

A basic message, one with no member data or user-defined methods still needs to provide typing information so that it can be properly forwarded to the desired node within a process instance. Consider a message construct with no explicitly defined parent type and no member data or methods, called generic. It is depicted in figure 7-1, with the relevant output in figure 7-2.

```
{ message:Generic; }
```

Figure 7-1 A simple message construct in *Generic.msg*

```
namespace sodl
{
 class Generic : public Message
 {
  public:
   Generic(process p);
   static void typeInit(mtype t);
   virtual Message& copy(void);
   virtual Message& copy(ulong);

  protected:
   Generic(ulong n, ulong i, mtype t);

  protected:
   virtual void instanceInit(void);
 };
}
```

(a) – *Generic.h*

```
namespace sodl
{
 Generic::Generic(process p)
  : Message(p.getNode(), p.getIndex(), SMT_Generic)
 {
  Generic::instanceInit();
  init();
 }

 Generic::Generic(ulong n, ulong i, mtype t)
  : Message(n, i, t)
 { Generic::instanceInit(); }

 void Generic::instanceInit(void) {}
 Message& copy(void)
  { return new Generic(*this); }
 Message& copy(ulong)
  { Message& rv = copy(); rv.setEngine(i); return rv; }

 void Generic::typeInit(mtype t)
 {
  msgTypes[t][SMT_Generic] = true;
  Message::typeInit(t);
 }
}
```

(b) – *Generic.cxx*

Figure 7-2 Relevant output derived from *Generic.msg*

Here there are no user defined methods or data members. The focus of the message is to provide type information that can be used in determining how any recipient processes will process instances of this message type.

The public class constructor ***Generic*(process *p*)** is used to create a ***Generic*** class instance, and not one of its subclasses. The parameter *p* is the process identifier of the message source. The call to the parent class constructor breaks up the source process handle into its component parts and adds the type information. From this, the root ***Message*** constructor sets the message source process handle and its type information, and derives message generation time. The constructor then calls ***Generic*::*instanceInit*** to perform member data initialization, and ***Generic*::*init*** to perform user-specified instance initialization. In this case, since there is no user-define init method, this results in calling ***Message*::*init***, which is empty.

Derived classes use the protected class constructor, ***Generic*(ulong *n*, ulong *i*, mtype *t*)**, to pass source and type information on to the ***Message*** class constructor. Like the public constructor, this constructor also calls ***Generic*::*instanceInit*** to perform local data member initialization. It does not call the user-specified initializer, since the language specification requires it to be explicitly called from the ***init*** method in the derived class being instantiated.

The ***Generic*::*copy*(void)** and ***Generic*::*copy*(ulong)** methods are used to produce copies of the message. The former returns a strict copy, while the other returns a copy but changes the message engine setting so that it can be controlled by an engine instance other than the one on which it was created.

Static method ***Generic*::*TypeInit*(mtype *t*)** is used to initialize the ***sodl*::*Defs*::*msgTypes*** array containing the parent-child relationship between different message types. This routine is called once during the simulation system setup, prior to any messages actually being delivered.

7.1.2. Message construct with user-defined methods and data members

User-defined methods within a message construct are placed into the resulting C++ class declaration in the obvious fashion. Given the message construct depicted in Figure 7-3, its data members and methods declaration are added to the C++ class in the fashion depicted in Figure 7-4.

The ***instanceInit*** method is really only used when there is a need to initialize an array to some preset value. For instance, in Figure 7-5, we declare the message construct **message:*send_vector*** and specify a default initial value for the array, an equation relating each component of the array to its index in the array.

```
{
        message:AddSubordinate
        {
                process:subordinates[];

                method:add(public; void; process:n;)
                { subordinates.push_back(n); }

                method:getTX(public; bool;)
                { return !subordinates.empty() && Message::getTX(); }

                method:size(public; ulong;)
                { return subordinates.size(); }
        }
}
```

Figure 7-3 ***AddSubordinate.msg*** **with one data member and three methods**

```
namespace sodl
{
 class AddSubordinate : public Message
 {
  public:
   std::vector< process > subordinates;

...

  public:
   virtual void add(process );
   virtual bool getTX(void);
   virtual ulong size(void);

 };
}
```

(a) – ***AddSubordinate.h***

```
namespace sodl
{

...

 void AddSubordinate::add(process n)
 { subordinates.push_back(n); }


 bool AddSubordinate::getTX(void)
 { return !subordinates.empty() && Message::getTX(); }


 ulong AddSubordinate::size(void)
 { return subordinates.size(); }
}
```

(b) – ***AddSubordinate.cxx***

Figure 7-4 C++ Files resulting from ***AddSubordinate.msg***

```
{
        message:send_array
        {
                long:x[100] ( (31*@%((long) sqrt(#))) );
        }
}
```

Figure 7-5 An initiali ed array as a data member in a message construct

When the developer specifies an initial value for a data member, the actual initialization normally takes place in the class constructor proper. However, since SODL provides some additional flexibility in initializing arrays, this needs to be accomplished in a routine where we can perform different computations for each element. This is the primary purpose of the ***instanceInit*** method. Figure 7-6 depicts the ***instanceInit*** method for the ***send_array*** class generated because of processing the code in Figure 7-5.

```
...
void send_array::instanceInit(void)
{
 for (long xindex=0; xindex<100; ++xindex)
   x.push_back( (31*xindex%((long) sqrt(100))) );
}
...
```

Figure 7-6 Initialization of the array specification in Figure 7-5

Note in particular the macro substitution of the '*xindex*' and '100' for the '@' and '#' respectively in the user-defined code.

7.2. Process construct files

Like messages, the C++ files generated from process constructs provide type information for the run-time system and declare user-defined data members and methods. In addition, process constructs provide a framework for processing messages.

7.2.1. A simple process construct

```
{process:Shape2D(Shape);}
```

Figure 7-7 A simple process construct, *Shape2D.proc*

The simplest process construct is one with no data members, methods or modes. An example of such a process is depicted in Figure 7-5 and the resulting C++ code is in Figure 7-6.

Like the class declaration resulting from a message construct, process constructs result in class declarations with two constructors. The first is a public constructor that is used to create an instance of that class, not a derived class. The one parameter in the public constructor is the index of the engine where the process will reside. The host engine is polled for its next available index which, when combined with the engine index, is used to create the unique handle identifying the newly created instance. This handle information and the type information is passed to the parent constructor all the way to the ***sodl::Process*** constructor which will create a new process controller, and perform some setup on the actual process instance. The second constructor is protected, and is intended can to be called from the constructor of a derived class. In both of these cases, the ***instanceInit*** method is called to perform some additional initialization. Unlike the C++ class declaration resulting from a message construct, the user-defined ***init*** method (if any) is not called at this point. The ***init*** method is called at the beginning of the simulation system initialization, but after all of the processes have actually been instantiated.

```
namespace sodl
{
 class Shape2D : public Shape
 {
  protected:
   Shape2D(ulong n, ulong i, ptype t);

  public:
   Shape2D(ulong n);
   static void typeInit(ptype t);
   virtual Process& copy(void);

  protected:
   virtual void instanceInit(void);
 };
}
```

(a) ***Shape2D.h***

```
namespace sodl
{
 Shape2D::Shape2D(ulong n)
  : Shape(n, nextProcess(n), SPT_Shape2D)
 { Shape2D::instanceInit(); }

 Shape2D::Shape2D(ulong n, ulong i, ptype t)
  : Shape(n, i, t)
 { Shape2D::instanceInit(); }

 void Shape2D::instanceInit(void)  {}
 Process& Shape2D::copy(void)
  { return new Shape2D(*(this); }

 void Shape2D::typeInit(ptype t)
 {
  procTypes[t][ SPT_Shape2D] = true;
  Shape::typeInit(t);
 } }
```

(b) ***Shape2D.cxx***

Figure 7-8 Relevant output from *Shape2D.proc*

The ***copy*** method is used to return a copy of the process instance. ***typeInit*** performs precisely the same function as the method by the same name in the message construct derived class declaration, except that it initializes the parent-child relationship between process types rather than message types.

7.2.2. Process constructs with data members and methods

Methods and non-process data members are handled identically to message constructs. For a more complete treatise on this aspect of C++ code generation for the SODL system, refer to section 7.1.2. However, a process can declare subordinate process instances within its definition. These process instance declarations are handled similarly to regular data members, but there are some differences, as depicted in Figures 7-9 and 7-10.

Note that the ***child***'s data type in the C++ class definition is type ***sodl*::process**. Also, note that the new ***sodl*::*Child*** instance associated with child is actually allocated in the class constructor. This has the default effect of creating a subordinate process on the same engine as the instance owning the ***sodl*::*Child***.

```
{import process {Child} }

{
        process:Simple
        {
                Child:child
        }
}
```

Figure 7-9 *Simple.proc* declaration of a subordinate process instance to process:*Simple*

```
namespace sodl
{
 class Simple : public Process
 {
  protected:
    process child;

  protected:
    Simple(ulong, ulong, ptype);

  public:
    Simple(ulong);
    static void typeInit(ptype);

  protected:
    virtual void instanceInit(void);
 };
}
```

(a) *Simple.h*

```
#include "Child.h"

namespace sodl
{
 Simple::Simple(ulong n)
  :       Process(n, nextProcess(n), SPT_Simple),
          child( (new Child(me.getNode()))→getID() )
 { Simple::instanceInit(); }

 Simple::Simple(ulong n, ulong i, ptype t)
  :       Process(n, i, t),
          child( (new Child(me.getNode()))→getID() )

 { Simple::instanceInit(); }
```

(b) *Simple.cxx*

Figure 7-10 C++ declaration and allocation of a subordinate process

```
{import process {ball} }

{
        process:bounce
        {
                ball:b[400]:1+(@%2);
        }
}
```

Figure 7-11 *bounce.proc* declaration of subordinate processes on non-default engines

While this method works well for single process instances in a given process definition, it does not easily extend to arrays of processes. A mechanism similar to array initialization described in section 7.1.2 is employed for this purpose and is depicted in Figures 7-11 and 7-12.

Note again the macro substitution in ***instanceInit*** of '***bindex***' for '@'. This splits the actual process instances across two engines, one on engine 1 and the other on engine 2.

```
namespace sodl
{
 class bounce : public Process
 {
  protected:
   std::vector<process> b;

  protected:
   bounce(ulong, ulong, ptype);

  public:
   bounce(ulong);
   static void typeInit(ptype);

  protected:
   virtual void instanceInit(void);
 };
}
```

(a) *bounce.h*

```
#include "ball.h"

namespace sodl
{
 bounce::bounce(ulong n)
   : Process(n, nextProcess(n), SPT_bounce)
  { bounce::instanceInit(); }

 bounce::bounce(ulong n, ulong i, ptype t)
   : Process(n, i, t)
  { bounce::instanceInit(); }

 void bounce::instanceInit(void)
 {
  for (long bindex=0; bindex<400; ++bindex)
    b.push_back( (new ball(1+(bindex%2)))→getID() );
 }
}
```

(b) *bounce.cxx*

Figure 7-12 Declaration and allocation of an array of subordinate processes

7.2.3. Mode and node declarations

The primary purpose of a process is to receive input messages, change its internal state, and to format and transmit outgoing messages in response to those input messages. Each of these steps must be performed somewhere in the C++ code produced. The developer explicitly defines the way state data is changed in response to an input message of a given type, as well as ensuring that outgoing messages have the proper payload. The remaining functions of actually directing an input message to the proper node or nodes and forwarding the output messages to the intended recipients is accomplished behind the scenes in the C++ code produced by parsing the SODL process files.

7.2.3.1. Specifying non-array output messages

```
{import std {<iostream>} }
{import message {Generic, StartSimulation} }
{
        process:Simple
        {
                mode:start
                {
                        node:proc[StartSimulation:strt][Generic:om=>(me;):(0.0)]
                                {start.setActive(false);}
                }

                mode:run
                {
                        node:proc[Generic:im][Generic:om=>(me;):(getTime()+1.0)]
                                { om.setTX(getTime() < 100.0); }
                }
        }
}
```

Figure 7-13 *Simple.proc* sample mode & subordinate nodes

```
namespace sodl
{
 class Simple : public Process
 {
  public:
   ProcessMode run;
   ProcessMode start;

  protected:
   Simple(ulong, ulong, ptype);

  public:
   Simple(ulong);
   virtual void receiver(Message& msg);
   virtual void setTime(double t);

  public:
   virtual void runproc(Generic in, Generic& out);
   virtual void startproc(StartSimulation strt, Generic& om);

  protected:
   virtual void setupSimplerunproc(Generic& in, Generic& out);
   virtual void setupSimplestartproc(StartSimulation& strt, Generic& om);
 };
}
```

Figure 7-14 *Simple.h* C++ relevant components of header file for *Simple.proc*

```
namespace sodl
{
...
 void Simple::setTime(double t)
 {
  Process::setTime(t);
  run.setTime(t);
  start.setTime(t);
 }

 void Simple::receiver(sodl::Message& msg)
 {
  if (run.isActive())
  {
   if (msg.isType(SMT_Generic))
   {
    Generic om(me);
    Generic& im = dynamic_cast<Generic&>(msg);
    runproc(im, om);
    setupSimplerunproc(im, om);
    getController().transmit(om);
   }

  }
  if (start.isActive())
  {
   if (msg.isType(SMT_StartSimulation))
   {
    Generic om(me);
    StartSimulation& strt = dynamic_cast<StartSimulation&>(msg);
    startproc(strt, om);
    setupSimplestartproc(strt, om);
    getController().transmit(om);
   }
  }
 }

 void Simple::runproc(Generic im, Generic& om) { om.setTX(++count < 100); }

 void Simple::startproc(StartSimulation strt, Generic& om)   { start.setActive(false); }

 void Simple::setupSimplerunproc(Generic& im, Generic& om)
 { if (!om.isPreempted()) om.addDest(me); }

 void Simple::setupSimplestartproc(StartSimulation& strt, Generic& om)
 {
  if (!om.isPreempted()) om.addDest(me);
  if (!om.timeOverride()) om.setTime(0.0);
 }
}
```

Figure 7-15 ***Simple.cxx*****, message handling implementation**

Figures 7-13 through 7-15 depict the C++ code generated that performs the functions associated with message handling within a process construct. We first note that each mode is declared as ***sodl::ProcessMode*** instances. The first portion of the message cycle is for each process to receive the message and check to see if any of the active nodes can handle it. This is performed in the receive method.

Upon receipt of an inbound message, the process controller calls the ***receiver*** method for the process. From there, each of the process modes is polled for its active status. All active modes will get a chance to process the incoming message, provided they have nodes capable of processing messages of the input message type. Upon entry into a code block representing an active mode, each node checks to see if it can accept a message of the given type or of a derived type. Upon finding a node that can actually handle the message, *msg* is cast to the proper type and output messages are declared. At this point, a properly typed copy of the input message and a collection of references to output messages are passed to the method given by the name *Mode-NameNode-Name* which contains the user specified code for manipulating the internal state data of the process, as well loading the output message payloads. This code is copied directly from the user code that was in the input process construct declaration.

Once the user-defined portion of handling the message has been performed, final output message setup is handled in method ***setup***_Process-NameMode-NameNodeName._ This involves adding any default destinations and time stamp values specified in the output message declaration. This method checks internal flag values to ensure that code in the user-defined portion of the message handling did not override these default values. If there was no override, the default destinations are added, and the message time stamp is set to its default value, if specified (if it was not specified, either in the node header or body, then Equation 6-1 is invoked).

The ***setTime*** method may at first glance seem to be out of place and unnecessary. However, when ***sodl::ProcessMode*** variables change their active flag, this change cannot take place immediately, since there may be additional pending messages with the same time stamp value as the current message being processed. Since the order of these identically time stamped messages is not defined, we require that they be processed in a well-defined manner regardless of the order in which they were actually received. Thus, any changes to the mode active flags cannot take effect until the process time stamp advances. By overloading the ***setTime*** method, we can set the time stamp value in the ***sodl::ProcessMode*** instances in a manner that allows consistent message processing.

7.2.3.2. Output message arrays

```
{import process {Vertex2D} }
{import message {SetVertex2D } }

{
        process:hierarchy
        {
                Vertex2D:vert[4];
                mode:Default
                {
                        node:start
                                [StartSimulation:in]
                                [SetVertex2D:out_sv[4]=>(vert[@];):(@*(-1e-9))] { ... }
}       }       }
```

Figure 7-16 *hierarchy.proc* with an output array of messages

```
void hierarchy::receiver(Message& msg)
{
 if (Default.isActive())
 {
  if (msg.isType(SMT_StartSimulation))
  {
   vector<SetVertex2D> out_sv;
   for (int out_sv_indexa = 0; out_sv_indexa < 4; ++out_sv_indexa) out_sv.push_back(me);

   StartSimulation& in = dynamic_cast<StartSimulation&>(msg);
   Defaultstart(in, out_sv);
   setuphierarchyDefaultstart(in, out_sv);

   for (int out_sv_indexb = 0; out_sv_indexb < out_sv.size(); ++out_sv_indexb)
    getController().transmit(out_sv[out_sv_indexb]); } } }

void hierarchy::Defaultstart(StartSimulation in, vector<SetVertex2D>& out_sv) { ... }

void hierarchy::setuphierarchyDefaultstart( StartSimulation& in, vector<SetVertex2D>& out_sv)
{
 for (ulong out_svindex=0; out_svindex<out_sv.size(); ++out_svindex)
 {
  if (!out_sv[out_svindex].isPreempted()) out_sv[out_svindex].addDest(vert[out_svindex]);
  if (!out_sv[out_svindex].timeOverride()) out_sv[out_svindex].setTime( (out_svindex*(-1e-9)) );
 } }
```

Figure 7-17 Relevant portions of *hierarchy.cxx* implementing message arrays

As with other type of arrays within the SODL language, certain aspects of output message arrays can be initialized using the macro substitutions described in Chapter 6. Figures 7-16 and 7-17 depict the C++ code resulting from processing a SODL construct with an output message array.

Here we again see the three relevant methods. The first is the receiver method sets up the output messages, though this time it is in the form of an array. This array along with the input message is passed to the user-defined message handler where the message array elements are manipulated. The messages are then passed to the setup routine to undergo final addressing and time stamping. Note again the macro substitution of '*out_svindex*' for the '@' in the original node declaration.

Chapter 8. GLUT-Based User Interface

8.1. Output concerns in an optimistic simulator

Optimistic simulation requires that all process states be recoverable in the event of a rollback. While this may be useful for actually performing the calculations associated with the simulation, it has its drawback in the vital area of I/O operations.

Output operations in particular are inherently irrevocable, and therefore must be handled differently from the actual simulation system. For simple console or file output, the SODL system provides the ***fossilCollect*** method. Suppose the $LP_i(t_s)$ process state is being fossil collected. Then t_s<GVT meaning that all remaining messages in the simulation system have time stamps strictly greater than t_s. Therefore, any output file or console output we would have liked to perform in $LP_i(t_s)$ can be safely performed during fossil collection without the risk of it ever becoming invalid.

The SODL graphics sub-system, known as the GLUT View Manager (GVM) uses a buffering mechanism to store pending changes to a scene graph. Those changes are made during fossil collection, and can be rolled back if the need to do so arises before actually being committed to the output device.

8.2. Overview of the SODL/GVM subsystem

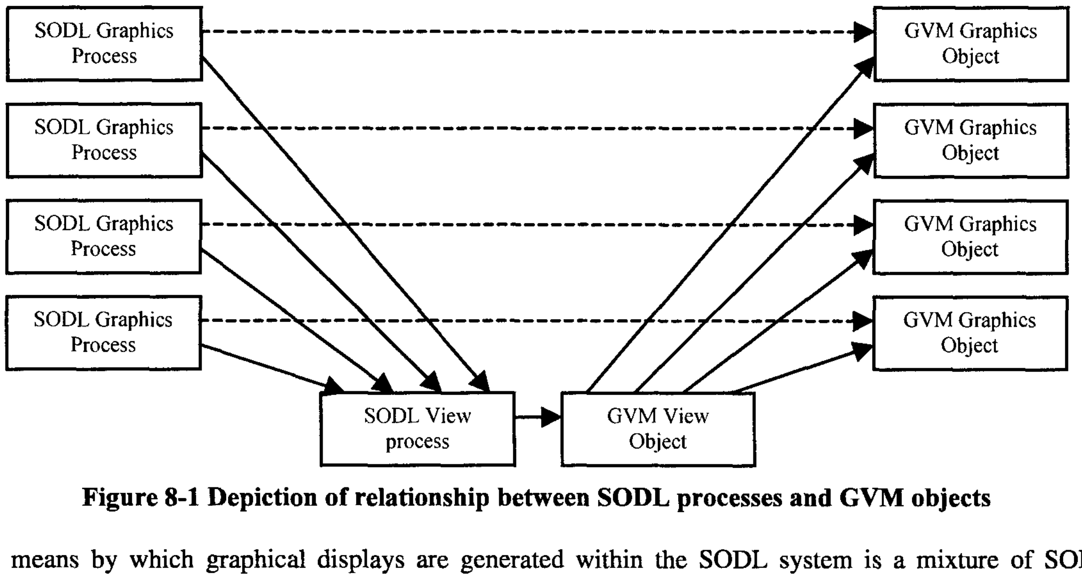


Figure 8-1 Depiction of relationship between SODL processes and GVM objects

The means by which graphical displays are generated within the SODL system is a mixture of SODL process and message constructs, and a collection of C++ classes that make up the GVM. The SODL process and GVM object instances can be thought of as mirror images of each other. SODL provides a collection of process and message constructs that allow programmers to specify their intent with regards to graphical output. This intent is then communicated to the GVM portion to actually implement those requests in a manner that is consistent with the notions of optimistic simulation, namely that output not occur until it is certain that it cannot be revoked.

Figure 8-1 depicts in somewhat more detail the notion of this SODL process-GVM object association. Each SODL process sends messages to any view processes in which it has been registered when it receives a request to change one of its parameters. The SODL view process then forwards to its GVM view object a request to update the actual GVM object responsible for rendering the object in the display. During fossil collection, this change is finalized. The next time the scene is actually drawn, the change is reflected on the screen.

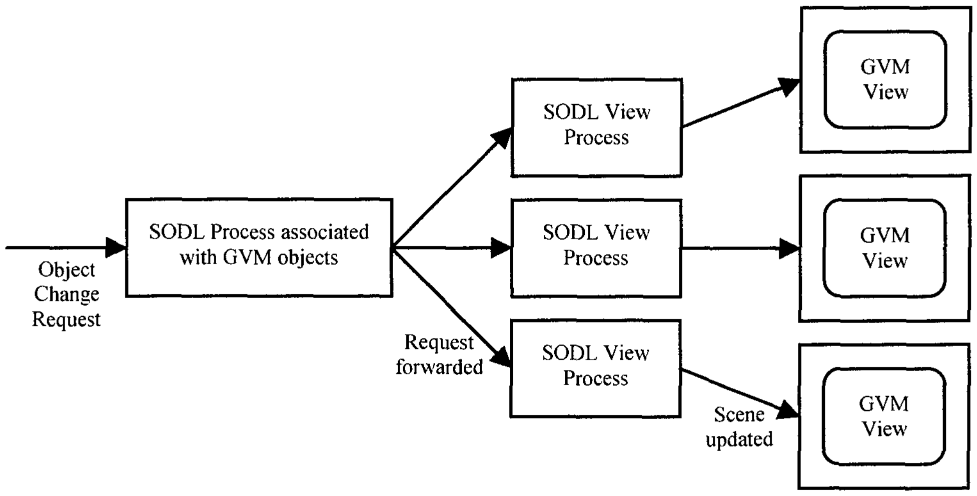


Figure 8-2 Message propagation for scene update requests

Figure 8-2 shows how updates are made to the scenes actually rendered. In order to see a change in the rendered scene, a message must be sent to the SODL process being changed corresponding to the GVM object being changed. This message is buffered and sorted according to its time stamp, possibly causing a rollback. When it is delivered, the receiving process will generate a message for each of the views in which it has a rendered object corresponding to itself to inform those views of the change. Those messages must also be transmitted, buffered and sorted before delivery to the view process responsible for a particular output display. Each view then schedules the change to occur in the actual rendered scene during fossil collection.

One drawback of this approach is that it is a bit slow. However, given the constraints of the SODL system, and ill-fated attempts at other approaches, this seemed the best way to solve the problem of providing a flexible output capability that could be, from the programmer's perspective, distributed across a collection of host machines.

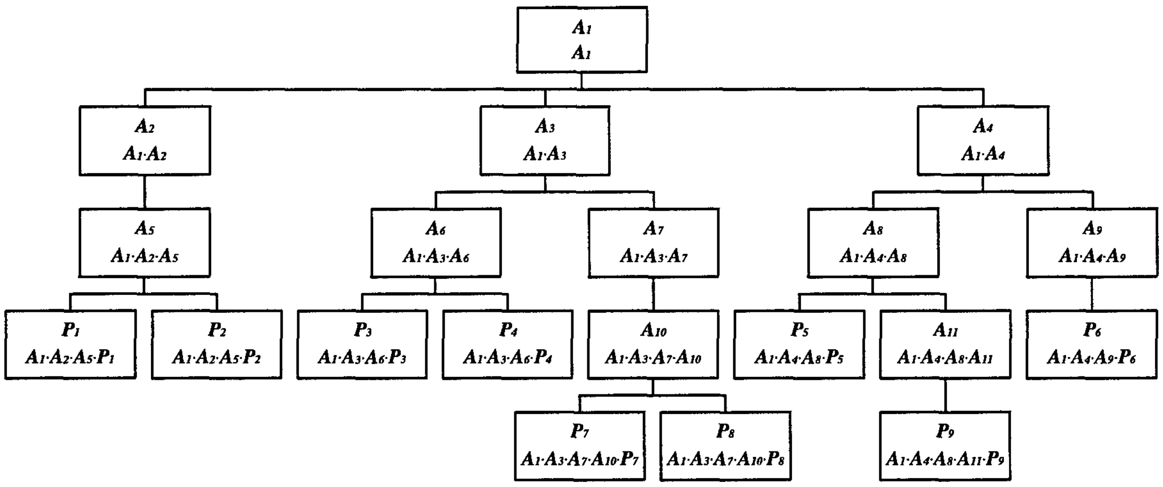
Although not implemented in the current SODL system release, this approach provides a way for users to communicate with processes corresponding to objects in the rendered scene. There is considerable work remaining to get this to happen, but it is a natural extension of that already done in this area.

8.2.1. SODL/GVM scene graphs

At the core of the GVM sub-system is the scene graph. The notion of a scene graph is described in more detail in (Foley 1996), but we will provide some basics here. In 2 and 3 dimensional graphics output, polygons are displayed to the screen, after having been manipulated by an affine transformation. These affine transformations are actually implemented as matrices that when multiplied together cause a succession of transformations to take place[11]. For example let $\boldsymbol{v}$ be a vector and let $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ be affine transformations. Then the process of applying $\boldsymbol{A}_1$ to $\boldsymbol{v}$ then $\boldsymbol{A}_2$ to that result can be expressed as the multiplication sequence $\boldsymbol{A}_2 \cdot \boldsymbol{A}_1 \cdot \boldsymbol{v}$. These affine transformations can perform a number of functions, but among the most common are translation, rotation, and scaling. When combined in different orders, these transformations can be used to manipulate polygon vertices in very complex manners.

The GVM scene graph can be thought of as a tree structure. Internal nodes are affine transformations, and leaves are polygons or other shapes. The affine transformation at any point in the graph is the product of the affine transformations in all of the ancestor nodes up to the root. When a given polygon is displayed to the screen, the affine transformation applied to its vertices is the product of all those transformations in the ancestor nodes.

An example of this relationship is depicted in Figure 8-3. Each internal node in the tree in Figure 8-3 represent an affine transformation labeled $\boldsymbol{A_x}$. This affine transformation is appended to the list of transformations from the parent nodes, and is explicitly shown as the second line in each of the internal nodes. Each vertex in polygon $\boldsymbol{P_y}$ is then multiplied by the aggregate affine transformation to get its final location in the rendered scene.



A_1 A_1
A_2 $A_1 \cdot A_2$
A_3 $A_1 \cdot A_3$
A_4 $A_1 \cdot A_4$
A_5 $A_1 \cdot A_2 \cdot A_5$
A_6 $A_1 \cdot A_3 \cdot A_6$
A_7 $A_1 \cdot A_3 \cdot A_7$
A_8 $A_1 \cdot A_4 \cdot A_8$
A_9 $A_1 \cdot A_4 \cdot A_9$
P_1 $A_1 \cdot A_2 \cdot A_5 \cdot P_1$
P_2 $A_1 \cdot A_2 \cdot A_5 \cdot P_2$
P_3 $A_1 \cdot A_3 \cdot A_6 \cdot P_3$
P_4 $A_1 \cdot A_3 \cdot A_6 \cdot P_4$
A_{10} $A_1 \cdot A_3 \cdot A_7 \cdot A_{10}$
P_5 $A_1 \cdot A_4 \cdot A_8 \cdot P_5$
A_{11} $A_1 \cdot A_4 \cdot A_8 \cdot A_{11}$
P_6 $A_1 \cdot A_4 \cdot A_9 \cdot P_6$
P_7 $A_1 \cdot A_3 \cdot A_7 \cdot A_{10} \cdot P_7$
P_8 $A_1 \cdot A_3 \cdot A_7 \cdot A_{10} \cdot P_8$
P_9 $A_1 \cdot A_4 \cdot A_8 \cdot A_{11} \cdot P_9$

Figure 8-3 Sample scene graph; Affine transformations are A_x, polygons are P_y

What this accomplishes is that objects displayed to the graphics device can have a hierarchy of sorts allowing individual components of a larger structure to be moved around with that larger structure. If the smaller components are capable of moving with respect to the larger structure, adjusting the affine transformation parameters affecting only the smaller component will cause such movements to appear in the final rendered scene.

The complication that GVM must contend with is that a simulation system may periodically send instructions to the GVM viewer to change some aspect of the scene graph. This implies that each of the objects associated with a specific view need to be able to receive messages, making them like processes. It might also be useful to be able to have a single SODL object correspond to objects displayed in multiple views of the same scene.[12] Therefore, a change in one affine transformation node could affect scenes

[11] Matrices are by definition linear transformations. It is possible, by adding an additional dimension to get them to mimic affine transformations in the lower dimension. A thorough description is available in any reference on computer graphics, but notably (Foley 1996)
[12] This would be particularly useful in a distributed implementation of the SODL system, allowing the scene to be displayed independently on different host computers in the distributed simulation.

rendered in more than one view. Since the intent was to provide a distributed simulation capability, direct manipulation of a view's scene graph cannot occur, since, in such distributed implementations, the scene graph may not reside locally.

What GVM has done is to have two separate but identical scene graphs. The first is the one that can send and receive messages, and is therefore actually implemented as a collection of SODL process and message files. Programmers making use of it send messages to these processes which forward requests to modify the scene graph in the actual view where the scene is rendered; this is the second scene graph. The SODL processes must retain all of the state information that is in the rendered scene, so that if a graphics object is added to a new view, it can inform that view as to its state and any subordinate objects it may own.

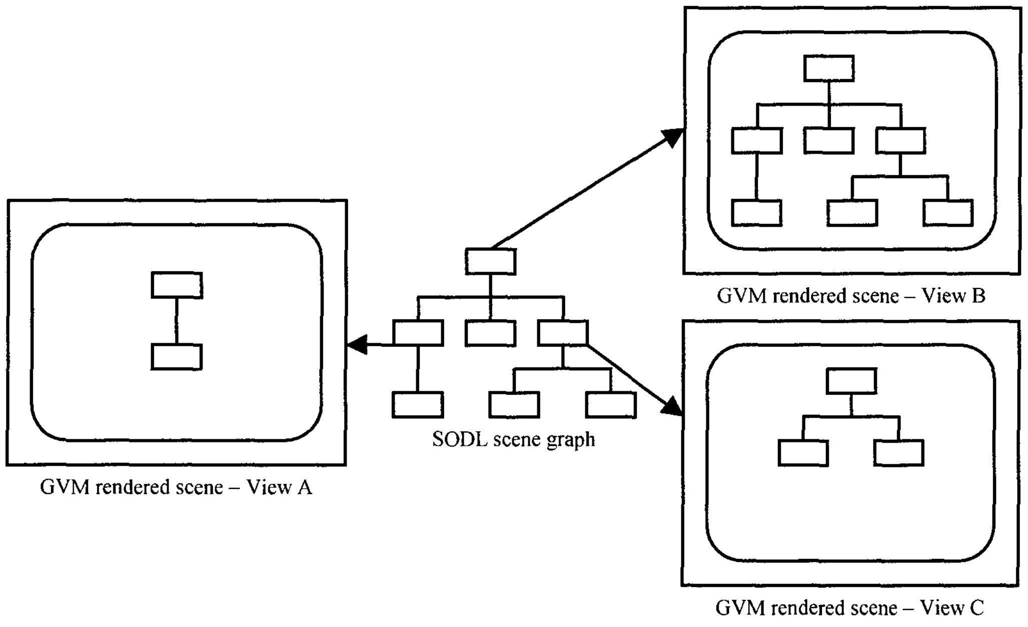

GVM rendered scene – View B
SODL scene graph
GVM rendered scene – View A
GVM rendered scene – View C

Figure 8-4 Portions of SODL scene graph displayed on multiple views

Figure 8-4 provides a pictorial representation of the GVM architecture. Depicted here is the SODL portion of the scene graph at the center of the figure. Portions of the SODL scene graph are replicated on different rendered scenes in different views. An update in one of the SODL objects will propagate to all of the scenes in which the object has corresponding rendered objects. The end result is that programmer does not need to manage the details of the individual scenes rendered, but can instead manage where different objects are viewed, and the relationship between the various rendered objects.

8.3. SODL/GVM usage

To actually make use of the SODL/GVM system, programmers need only declare in their process constructs a **process:*View*** derived instance, along with the affine transformation nodes and shapes to actually display.

There are two types of **process:*View*** constructs defined, **process:*View2D*** for displaying two-dimensional information and **process:*View3D*** for three-dimensional displays. Programmers should use either of these two process constructs, or derive new ones based upon their specific requirements. The view can be considered the root node, and contains the affine transformation for the view orientation and position. This viewing transformation is then applied to all subordinate nodes.

Each view can have a collection of subordinate nodes. These are either **process:*Node2D*** or **process:*Node3D*** instances, as appropriate. The only real difference between them is the size of the various arrays they have governing the affine transformations they represent. They have the ability to govern translation, scaling, and rotation. The center of rotation and center of scaling can also be specified. To add a node with process handle (N_n, I_n) to a view with process handle (N_v, I_v), a user sends a **message:*AddNode2D*** or **message:*AddNode3D*** (both derived from the **message:*AddNode*** construct) as appropriate, to (N_v, I_v). The **message:*AddNode*** derived messages have the ability to add multiple nodes. To load the message, call its **method:*add*(public; void; process:*p*)** method to add a ***sodl*::process** value to the list of them. These added values are the process handles of the nodes we would like to add as subordinates to the destination.

A **process:*Node*** instance can reside on multiple views. To do this, simply send one of the **message:*AddNode*** derived constructs to each view instance where the **process:*Node*** is intended to reside.

The system provides the capability to, not only add **process:*Node*** instances to a **process:*View***, but also to other **process:*Node*** instances. We can use the **message:*AddNode*** derived constructs for this as well, though we send it to the **process:*Node*** construct instance that will be getting the new subordinate **process:*Node*** instances. Any view in which the receiving node has been registered will automatically be informed about the addition of the new subordinate.

```
{import process {View3D, Node3D, Dodecahedron} }
{import message {AddNode3D, AddShape3D, StartSimulation} }

{
        process:dode
        {
                View3D:view;            // Display to this view
                Node3D:root;            // Node for the display
                Dodecahedron:shape;     // A dodecahedron to view

                mode:Default
                {
                        node:start_sim   [StartSimulation:strt]              // Bootstrap message
                                         [AddNode3D:an=>(view;),              // Add root to view
                                         AddShape3D:as=>(root;)]              // Add shape to root
                        {
                                an.add(root);      // Set the root node as a subordinate of the view
                                as.add(shape);     // Set the shape as a subordinate of the root node
                        }
                }
        }
}
```

Figure 8-5 dode.proc, a simple view with a single affine transformation node and shape

There are several predefined shapes for the three-dimensional views. They include **process:*Cone***, **process:*Cube***, **process:*Dodecahedron***, **process:*Icosahedron***, **process:*Octahedron***, **process:*Polygon3D***, **process:*Sphere***, **process:*Tetrahedron***, and **process:*Torus***. All of these are derived from the **process:*Shape3D*** construct. With the exception of the **process:*Polygon3D***, all of them are fixed in terms of their vertices and edges. The **process:*Polygon3D*** construct can have subordinate **process:*Vertex3D*** constructs, the position of which can be changed via message passing. There are some limitations to how these vertices may be used. Specifically, in certain OpenGL drawing modes, the locations of all of the vertices must be coplanar and form a convex polygon. Since all polygons can be decomposed into a collection of convex polygons, this is more of a nuisance than a real limitation of the system. For two-dimensional views, the only predefined construct is the **process:*Polygon2D***, which has the same limitations as the **process:*Polygon3D***, and restricts ownership to only **process:*Vertex2D***.

To add a shape to a **process:*Node***, send the appropriate **message:*AddShape*** derived construct (either **message:*AddShape2D*** or **message:*AddShape3D***) to the node to which the shapes will be subordinated. Again, the addition of this new shape will be forwarded to all views in which the node has been registered.

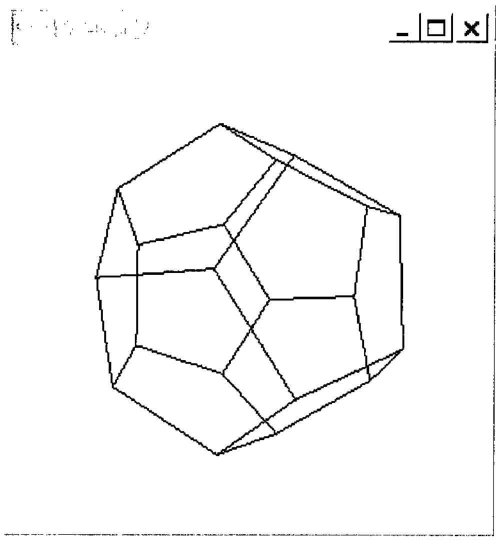


Figure 8-6 Output of dode.proc

Figure 8-5 has code for displaying a simple geometric shape in a **process:*View3D*** window. Figure 8-6 shows the output for the program in Figure 8-5.

8.4. SODL/GVM Architecture

Detailed descriptions of both the SODL and GVM portions of the display subsystem are documented in Appendix B, sections 3 and 4 respectively. We provide a somewhat broader overview here to help programmers wishing to use the system to do so effectively.

As stated earlier, for each SODL process used to control display content, there is an associated GVM class to actually manipulate and display the scene graph. The programmer only needs to send messages to the SODL side of the system. The updates to the displayed scene are handled at first within the SODL system, and then passed to the GVM portion where these update requests are buffered until fossil collection. During fossil collection, the requests are used to actually perform updates to the scene.

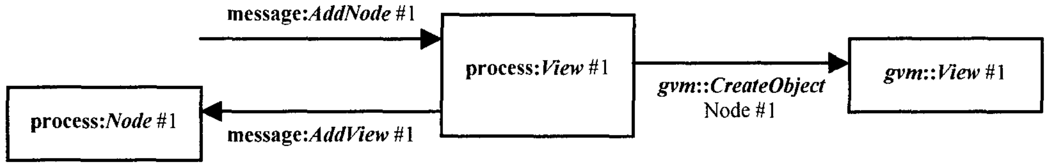
8.4.1. SODL view controllers, the process:View

From the SODL perspective, a single **process:*View*** instance controls exactly one GLUT view port. All objects displayed in the window, down to the individual vertices making up polygons, must be registered with the **process:*View*** instance associated with the GLUT window displaying them. All such objects are derived from the **process:*Object*** construct. This registration takes two forms. The first is to have a **process:*Node*** instance be added to a view's list of root nodes via the messages derived from **message:*AddNode***. The second is to have a parent object be registered with a new view, in which case all subordinate objects in the scene graph are also registered.

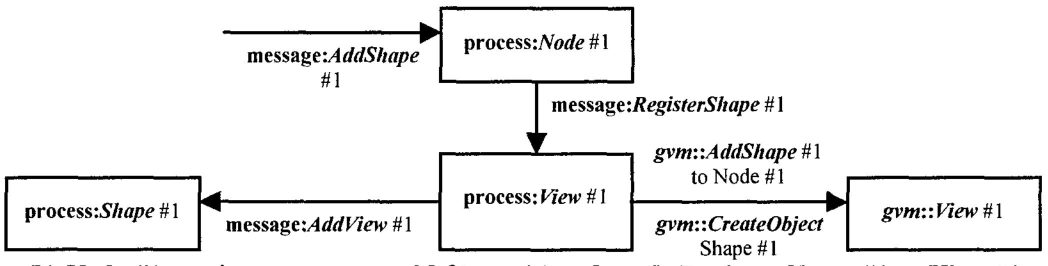
SODL views come in two flavors, **process:*View2D*** and **process:*View3D*** instances. Programmers should not directly instantiate a **process:*View***, since much of the functionality associated with manipulating views is implemented only in these two derived classes. The **process:*Node*** instances should be instantiated as either **process:*Node2D*** or **process:*Node3D*** for essentially the same reason. **process:*View2D*** and **process:*Node2D*** instances are used for controlling two-dimensional display data, while **process:*View3D*** and **process:*Node3D*** instances are for three-dimensional displays. The reason for this separation is that OpenGL can perform some optimizations for 2D displays making display of such data somewhat faster than it would otherwise be.

Figure 8-7 depicts how this registration is actually performed. The first step in the process comes when a request is made of a **process:*View*** instance to add a new root node to its list of them. The view maintains a list of known process handles, and checks to see if the handle for this **process:*Node*** instance is among them. If not, the **process:*View*** schedules the creation of a corresponding ***gvm*::*Node*** instance with the ***gvm*::*View*** instance it maintains. This returns a **gvm_index** value that can be used to identify that specific ***gvm*::*Node*** instance in the future. With this **gvm_index** in hand, the **process:*View*** sends a **message:*AddView*** to the node it has just added. Included in the payload of the message is the **gvm_index** value. Any messages the **process:*Node*** sends to that specific view must contain the proper **gvm_index** value so that instructions can be properly forwarded to the node's counterpart in the rendered scene graph. Therefore, the node maintains an associative memory with the handle for the **process:*View*** as a key and the **gvm_index** as its value and uses the index in any future communications with that view. Not depicted in

Figure 8-7 is the message **process:*Node*** sends back to the view. These messages notify the view as to the current state values of the node so that the proper values are relayed to the ***gvm::Node*** instance when it is created.



(a) View #1 receives request to add Node #1 as a root node. Sends AddView #1 to Node #1 and schedules creation of a corresponding node with its *gvm::View* instance



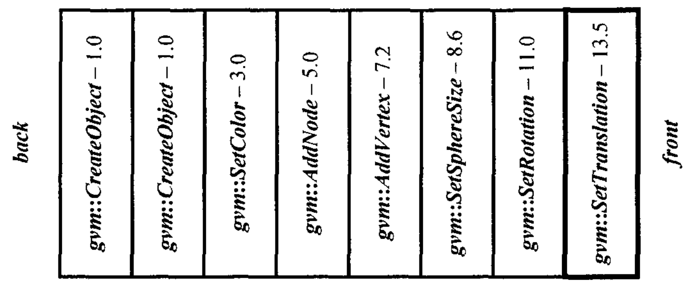
(b) Node #1 receives request to add Shape #1 and sends Register Shape #1 to View #1 which schedules with its *gvm::View* creation of a corresponding shape and establishment of subordinate relationship; sends an Add View #1 message to Shape #1

Figure 8-7 SODL side messaging

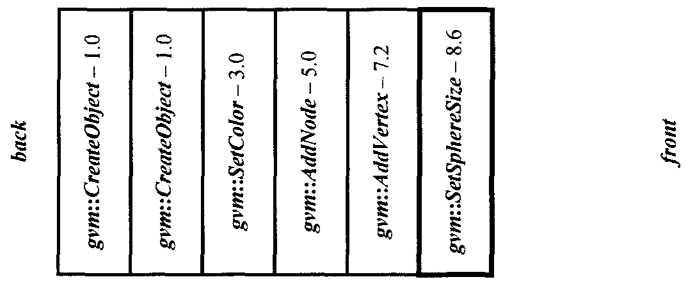
When a new shape is added to the list of subordinate objects for the node, essentially the same procedure is repeated. This time, the node sends a **message:*RegisterShape*** to the **process:*View*** instance along with the process handle of the shape to add. Again, the **process:*View*** will check its list of known processes, and if the handle is not yet registered, it will schedule a ***gvm::CreateObject*** event with its ***gvm::View*** to actually create the new shape instance. It also will schedule an update in the scene graph to add the newly created ***gvm::Shape*** instance as a subordinate shape to the ***gvm::Node*** instance it created earlier. The **process:*View*** instance will then send its **message:*AddView*** to the new shape, along with the **gvm_index** returned from scheduling the ***gvm::CreateObject***. The shape retains the **gvm_index** for use in any future communications with the view. Again, not pictured is the phase where the newly registered shape relays back to the **process:*View*** its state data so that additional ***gvm::Message*** instance can be scheduled with the ***gvm::View*** to provide current state information to the scene graph.

The same basic idea is used any time the state of a **process:*Object*** derived construct instance changes its state. It will forward a message to the views with which it has been registered, along with the **gvm_index** value for that view, notifying it of the change in its state. The **process:*View*** will schedule messages with the ***gvm*::*View*** notifying it of the change in the scene graph state.

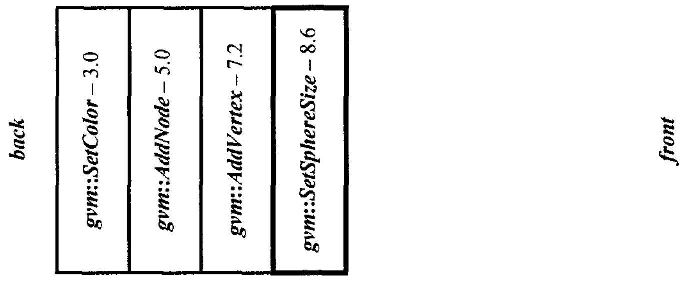
8.4.2. Messaging on the GVM side



(a) At LVT 13.5, the *gvm*::*SetTranslation* event was scheduled in the *gvm*::*View* instance by adding the event to the front of the deque.



(b) A rollback occurred restoring the state with time stamp 8.6. Events schedule for 11.0 and 13.5 were removed from the deque.



(c) Fossil collection for time 1.0 was performed. The specified objects were created and the scheduled events were removed from the back of the deque.

Figure 8-8 Scheduling, revoking and processing messages in the GVM message queue

When a **process:*View*** construct instance receives a message that one of the **process:*Object*** instance registered with it has had a state change, or when new **process:*Object*** instances are being initially registered with the **process:*View*** instance, these changes must be reflected in the scene graphs that are used to actually generate imagery to graphics output devices. However, each of these requests must occur in the proper time stamp order, and, since there is no guarantee that they are actually received in the proper order until fossil collection can be performed, it is only at that time that these changes to the scene graph can actually be committed.

There are three phases to the GVM side of the graphics subsystem as depicted in Figure 8-8.

1. *Scheduling*. All events are time stamped and, by virtue of the optimistic simulation mechanism the SODL system uses, are inserted into the front of a double-ended queue (also called a deque) in time stamp order.

2. *Rollback*. Since the messages are inserted into the deque in time stamp order, the messages can be removed from the front of the deque until the front most element has a time stamp not greater than the rollback time. Any events scheduled for a later time will, by definition be of a later time stamp value, thus the chronology of scheduled scene graph adjustments remains intact.

3. *Fossil collection*. All events with time stamps at or before the fossil collection time can be processed, allowing the scene graph updates to be made without the possibility of a rollback making those changes invalid.

Chapter 9. SODL Sample Programs

The main advantage of the SODL system is that it provides developers with the means to quickly and succinctly define the behavior of simulation objects without being bogged down with the details normally associated with simulation systems. As a means of providing some insight into how the SODL system works, and how it may be used to rapidly develop complex simulation systems, this chapter will explain in detail how the various demonstrations that come with the SODL distribution work and what they are intended to do.

Many of these samples produce output, but for brevity, the code segments listed here generally do not show how this out is actually produced. Complete listings of all of the demos and support code (if necessary) are provided in Appendix C.

9.1. Single Node Textual Simulations

These simulation are early demonstrations intended to test various aspects of the SODL system when all the simulation objects residing on the same engine. There is, therefore, no chance of ever getting a rollback, and the fossil collection is trivially completed. They are not intended to simulate anything in particular, just provide some basic samples for testing and illustration purposes.

9.1.1. Simple1

The Simple1 simulation, depicted in Figure 9-1, with constructs listed in Figure 9-2.

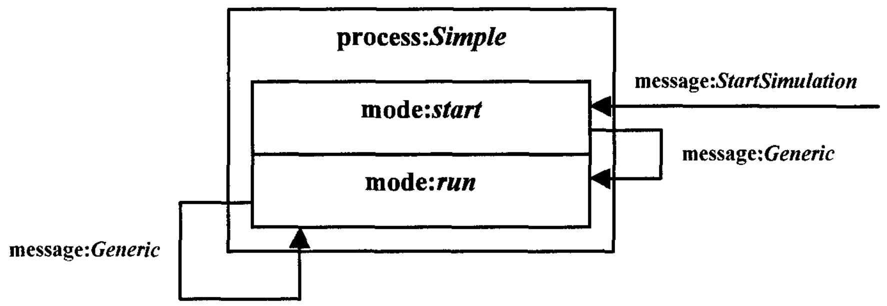


Figure 9-1 Schematic of Simple1

The purpose of this simulation was to test the basic message passing mechanism in the simplest possible configuration. A **process:*Simple*** instance upon start up (i.e. receipt of the **message:*StartSimulation*** instance at time -1) will send a **message:*Generic*** with time stamp 0.0 to itself. Every time it receives a **message:*Generic***, it transmits another one to itself, and increments a counter. Upon receiving 100 of these **message:*Generic*** instances, the 101st instance has its transmission flag turned off, preventing actual transmission of the message.

```
{ message:Generic; }
```

(a) Generic.msg

```
{import message {Generic, StartSimulation} }
{import std {<iostream>} }

{
        process:Simple
        {
                int:count(-1);                          // Hit counter

                method:init(public; void;) { std::cout.precision(15); }

                method:fossilCollect(public; void;)
                {
                        if (getTime() >= 0)             // If this is a good time to produce output?
                        {
                                if (getTime() == 0) std::cout << me << ": starting";
                                else std::cout << me << ": " << count << " @ time " << getTime();
                                std::cout << std::endl;
                        }
                }

                mode:start
                {
                        node:proc[StartSimulation:strt][Generic:om=>(me;):(0.0)]
                                { start.setActive(false); } }

                mode:run
                {
                        node:proc[Generic:im][Generic:om=>(me;)]
                                { om.setTX(++count < 100); } } } }
```

(b) Simple.proc

Figure 9-2 The Simple1 Simulation

The process starts out with all of the modes active. After the initial bootstrapping message is received, **mode:*start*** is deactivated so that the nodes within it are no longer polled for the remaining messages. All remaining messages are handled in **mode:*run***.

After each message is received, during the fossil collection phase, the process produces textual output to the screen to inform the user of the status of the simulation. Since there is only one process, and the possibility of a rollback is exactly 0, this output could have been handled in the node receiving the **message:*Generic*** instance. It is, instead placed in the **method:*fossilCollect*** to maintain the intent that output only be performed during fossil collection.

9.1.2. Simple2

Simple2 is in essence the same as Simple1, but was developed to test for memory leaks in the basic scenario. Its primary difference is that it does not end until the user manually terminates the run. To limit the amount of output sent to the screen, it produces output once in 10,000 iterations, rather than after each iteration as in Simple1.

The functionality of Simple2 is depicted, like that of Simple1, in Figure 9-1, and it uses the same definition of **message:*Generic***, depicted in figure 9-2. The code running the process is depicted in Figure 9-3.

```
{import message {StartSimulation, Generic} }
{import std {<iostream>} }
{
        process:Simple
        {
                long:count(-1);                         // Counter
...
                mode:start
                {
                        node:proc[StartSimulation:in][Generic:out=>(me;):(0.0)]
                                { start.setActive(false); } }

                mode:run
                {
                        node:proc[Generic:in][Generic:out=>(me;)] { count++; } } } }
```

Figure 9-3 Process construct for the Simple2 simulator

9.1.3. Simple3

The Simple3 demo is an example of a two-process simulation, and is depicted in Figure 9-4, with code displayed in Figure 9-5. The **process:*Simple*** declares a **process:*Child*** instance. Upon bootstrapping and some initialization to inform the child process who its parent is, **message:*Generic*** instances are passed back and forth between the two process instances ad infinitum.

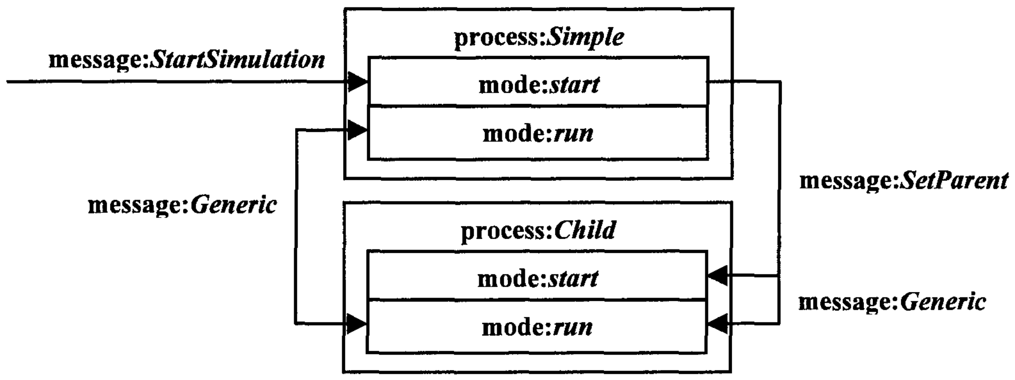


Figure 9-4 Message transport in Simple3

```
{import message {StartSimulation, Generic, SetParent} }
{import process {Child} }
{
        process:Simple
        {
                long:count(-1);                 // Counter
                Child:child;                    // Child process
...
                mode:start
                {
                        node:start[StartSimulation:strt]
                                [Generic:out=>(child;):(0.0), SetParent:sp=>(child;)]
                                { start.setActive(false); } }

                mode:run { node:bounce[Generic:in][Generic:out=>(child;)] { count++; } } } }
```

(a) Simple.proc

```
{import message {SetParent, Generic} }
{
        process:Child
        {
                process:parent;                         // Handle to the parent process
                long:count(-1);                         // Counter
...
                mode:start
                {
                        node:setParent[SetParent:in][]
                        {
                                parent = in.getSource();        // Set the parent reference
                                start.setActive(false); } }     // Turn this mode off
                mode:run
                {
                        node:bounce[Generic:in][Generic:out=>(parent;)] { count++; } } } }
```

(b) Child.proc

Figure 9-5 Simple3 process construct declarations

The **message:*Generic*** is as defined in Figure 9-2, and **message:*SetParent*** is essentially the same, though its type setting ensures that when it is delivered to the **process:*Child*** instance, the sender can be retrieved and saved for future reference.

9.1.4. Ping

```
{import message {Generic, StartSimulation} }
{import process {Pong} }
{
        process:Ping
        {
                int:count(-1);              // Count of the number of messages received
                Pong:pong;                  // Something to send a message to

...

                mode:start
                {
                        node:start[StartSimulation:strt] [Generic:out=>(me;):(0.0)]
                        { start.setActive(false); } }

                mode:run
                {
                        node:ponger[Generic:in][Generic:out=>(pong;)]
                                { out.setTX( ++count < 20 ); } } } }
```

(a) Ping.proc

```
{import message {Generic} }
{
        process:Pong
        {
                int:count(-1);              // How many times have we received a message
...
                mode:Default
                {
                        node:pinger[Generic:in][Generic:out=>(in.getSource();)]
                                { out.setTX(++count<20); } } } }
```

(b) Pong.proc

Figure 9-6 Source code for the Ping demo

The Ping demo is similar to Simple3. Figure 9-7 depicts how messages are passed between processes, and Figure 9-6 is the code for the Ping simulator. The main difference is that instead of retaining a reference to the sending process of **message:*Generic*** instances, the follow-on message is simply returned to the sender. As such, there is no **message:*SetParent*** to inform the subordinate process of its parent as there is in Simple3. In addition, after each process has received 20 **message:*Generic*** instances, the simulation stops automatically. This was an early demonstration to test the basic message passing capability of the SODL system.

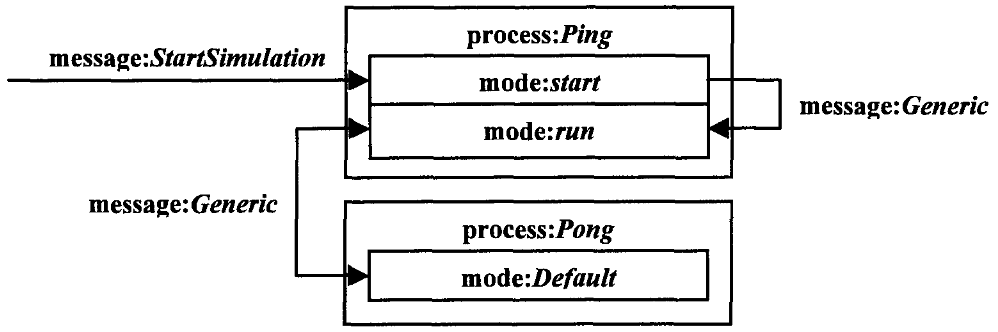


Figure 9-7 Ping message transport

9.1.5. Ring1

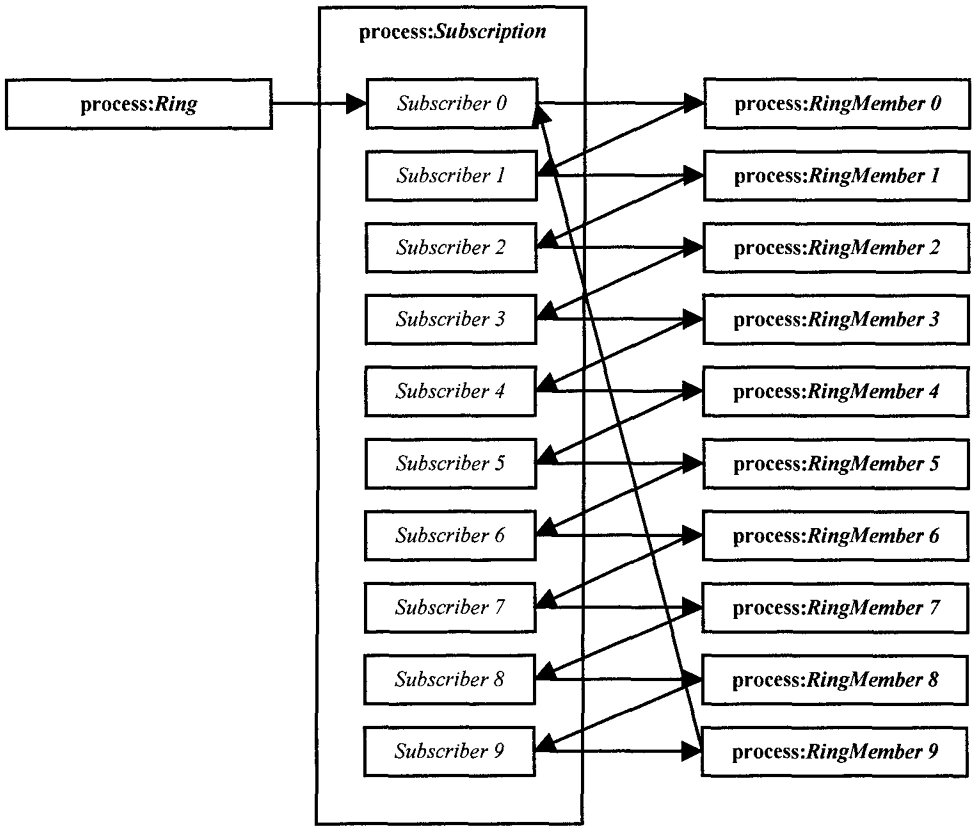


Figure 9-8 Ring1 Token ring using a subscription service

One of the original intentions of the SODL project was to provide a subscription service, whereby processes could sign up with a subscription and any message sent to the subscription would automatically be forwarded to its membership list. In the end, this proved to be too cumbersome to adequately implement in conjunction with the time warp algorithm, and this feature was dropped from the final system.

Ring1, and Ring2 are two demos that originally used these subscriptions. They have been re-implemented, this time with the subscription defined as a process. Ring1 consists of one controller process, called **process:*Ring***, a subscription process called **process:*Subscription***, and 10 ring members, declared in the **process:*RingMember*** construct.

During bootstrapping and initialization in the Ring1 demo, each ring member is informed as to the process handle of the subscription process. With this handle available, the ring members then request to subscribe to the subscription. As the subscription receives each request, it replies with a subscription index. Messages intended to be forwarded by the subscription can either be sent to individual subscribers, by specifying the index in the incoming message, or to all subscribers, when the recipient index is not specified.

```
{import process {RingMember, Subscription} }
{import message {Generic, StartSimulation, Setup, ReportSize} }

{
        process:Ring
        {
                RingMember:ring[10];              // Ring of elements
                Subscription:sub;                 // Subscription list

                mode:Default
                {
                        node:start[StartSimulation:strt]
                                [Setup:s=>(ring;),
                                 ReportSize:r=>(sub;):(-0.5),
                                 Generic:out=>(ring;):(0.0)]
                                { out.set(0); s.set(sub); } } } }
```

Figure 9-9 Ring.proc; The parent process for the Ring1 simulation sample

Once this portion of the setup is completed, the **process:*Ring*** instance requests that the subscription provide the subscriber count to all of the subscribers. Once this is completed, the real simulation starts.

```
{import message {Subscribe, ReportSize, ReportIndex, Generic} }

{
        process:Subscription
        {
                process:subscribers[];                    // List of subscribers

                mode:Default
                {       node:subscribe[Subscribe:in][ReportIndex:out=>(in.getSource();)]
                        {       out.set(subscribers.size()); // Index value to report back
                                subscribers.push_back(in.getSource());}

                        node:reportSize[ReportSize:in][ReportSize:out=>(subscribers;)]
                        { out.set(subscribers.size()); }

                        node:forward[Generic:in][Generic:out]
                        {       if (in.get() < subscribers.size())
                                        out.addDest(subscribers[in.get()]);
                                else
                                        out.addDest(subscribers);
                                out.set(in.get()); } } } }
```

(a) Subscription.proc

```
{import message {Generic, Setup, ReportSize, ReportIndex, Subscribe} }

{
        process:RingMember
        {
                process:sub;        // Handle to the subscription process
                long:count(-1);     // A simple counter
                long:next(0);       // Index of next instance
                long:index(0);      // Index of this instance

...

                mode:Default
                {
                        node:setup[Setup:in][Subscribe:out=>(sub;)] { sub=in.get(); }
                        node:setIndex[ReportIndex:in][] { index=in.get(); }
                        node:setNext[ReportSize:in][] { next = (index+1) % in.get(); }

                        node:run[Generic:in][Generic:out => (sub;) ]
                        {
                                out.set(next);      // Set index of the eventual destination
                                out.setTX(count++<10); } } } }
```

(b) RingMember.proc

Figure 9-10 Processes controlling (a) the subscription service and (b) individual ring members.

The notion of Ring1 is to perform a token ring simulation whereby a message is sent to the first element in the ring, which sends a message to the second, and so on until it gets to the last element. This last element will then forward a message back to the first. Rather than relying upon storing the handle of the next process, each **process:*RingMember*** instance instead retains its index so it can figure out which subscriber is next to get the message. The token, in this case another **message:*Generic*** instance is forwarded to successive elements until it has made ten complete circuits of the ring. This final phase of the simulation is depicted in Figures 9-8. Figures 9-9 and 9-10 contain the source code for the process constructs in the Ring1 demonstration.

The result is that each of the **process:*RingMember*** instances takes a turn receiving the token and passing it to the next subscriber. No two ring members have a token at the same simulation time, so the sequence of token passing is sequential, from one ring member to the next.

9.1.6. Ring2

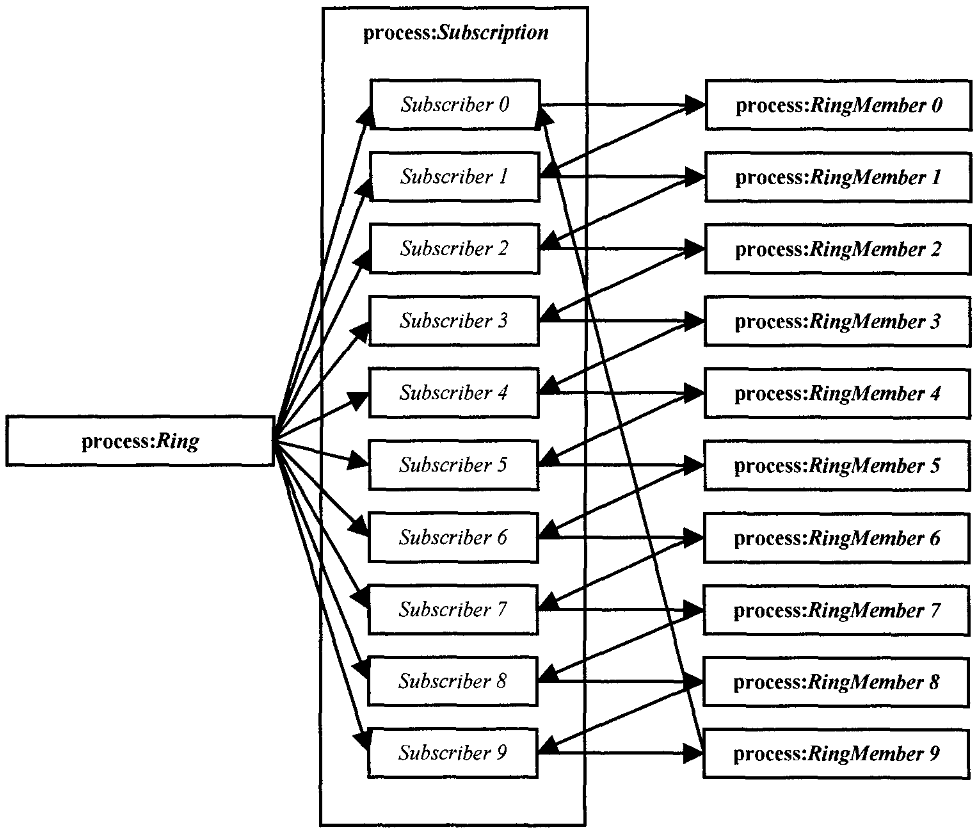


Figure 9-11 Message routing for Ring2 simulation

Ring2 is essentially the same as Ring1, except with regards to one minor change in the **process:*Ring*** construct declaration. This change broadcasts the initial **message:*Generic*** instance to all of the subscribers listed with the **process:*Subscription*** instance.

Figure 9-11 depicts the message flow following the setup portion of the simulation. Figure 9-12 shows the code change in Ring.proc to result in this change. Note that we removed the ***out.set***(0) in **node:*start*** of Figure 9-10. This directs the subscription to broadcast the **message:*Generic*** instance to all of its subscribers.

```
{import process {RingMember, Subscription} }
{import message {Generic, StartSimulation, Setup, ReportSize} }

{
        process:Ring
        {
                RingMember:ring[10];            // Ring of elements
                Subscription:sub;               // Subscription list

                mode:Default
                {
                        node:start[StartSimulation:strt]
                                [Setup:s=>(ring;),
                                 ReportSize:r=>(sub;):(-0.5),
                                 Generic:out=>(ring;):(0.0)]
                                { s.set(sub); } } } }
```

Figure 9-12 Ring.proc; The parent process for the Ring1 simulation sample

9.1.7. Brigade2

The Brigade2 simulates a brigade of soldiers performing some unspecified task. The brigade is broken into four battalions, which are in turn broken into four companies each. Each company is broken into four platoons, and these platoons are each broken into four squads. Finally, each squad is composed of ten soldiers. A quick computation reveals that there are, 2,901 units the simulation system must managed. This hierarchy is illustrated in Figure 9-13.

During a setup phase, each unit sends a message to each of its subordinate units to establish with that subordinate that the owning unit is its parent. Communication occurs only from one level to the next. That is, a unit can communicate only with its parent unit and its immediate subordinates.

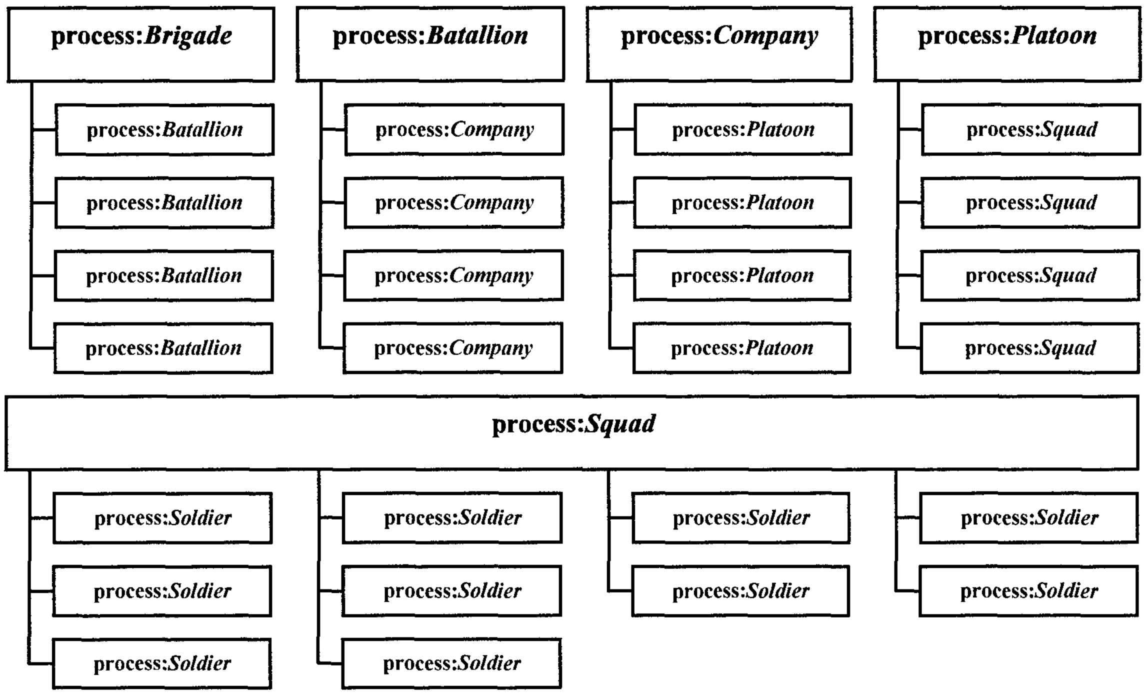


Figure 9-13 Process ownership in Brigade2 demonstration

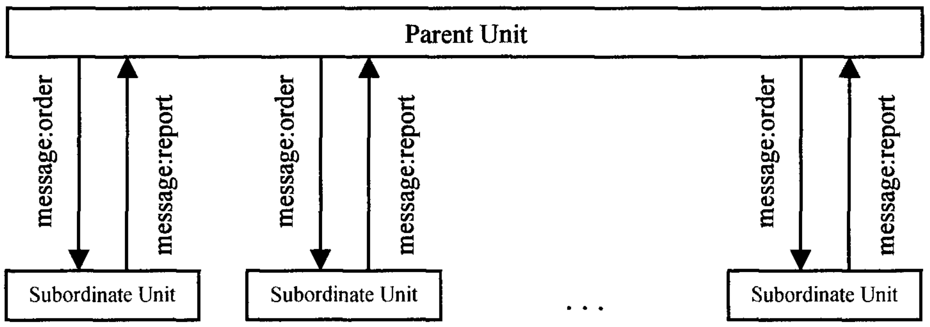


Figure 9-14 Communication between parent and subordinate units in Brigade2 demo

When the simulation actually gets under way, the brigade issues an order with different randomly determined time stamps to each of its subordinate battalions. These battalions then issue orders to each of their subordinate companies, again a random time stamps. This continues all the way down to the individual soldier units. After a period of time, the soldier will report to its parent unit that it has completed its task. When all of the soldiers in a particular squad have informed it that they have completed their assigned tasks, the squad will report this back to its parent platoon, and so on. The simulation ends when the brigade has completed its task. This is illustrated in Figure 9-14.

The bulk of the work is done in the **process:*Unit*** construct. All of the process constructs in the Brigade2 demo are derived from **process:*Unit***, which is summarized in Figure 9-15. **process:*Soldier*** is shown in

Figure 9-16 since it is fundamentally different from other units, in that it does not have any subordinate units.

```
{import message {report, order, set_parent, StartSimulation} }
{
        process:unit
        {
                int:instance;                   // Subordinate instance of this unit
                int:sub_count;                  // # of subordinates Squad/soldier needs to change to 10/1
                process:parent;                 // Parent unit
                process:subs[];                 // Handles to subordinate units
                double:sub_times[];             // Timestamp for subordinate units

                mode:start
                {
                        node:setParent[set_parent:in][set_parent:out[]=>(subs[@];)]
                        {
                                // Perform unit & subordinate initialization
                                start.setActive(false);                 // Deactivate start mode
                                waiting_for_orders.setActive(true); // Activate waiting_for_orders
                        } }

                mode:waiting_for_orders
                {
                        node:startSimulation[StartSimulation:in][]
                                { waiting_for_orders.setActive(false); }

                        node:receive[order:ord][order:out[]=>(subs[@];):(sub_times[@])]
                        {
                                // Non-soldier instances configure orders to subordinates
                                waiting_for_orders.setActive(false);            // Not waiting
                                working.setActive(true); } }                    // We are now working
                mode:working
                {
                        node:startSimulation[StartSimulation:in][] { working.setActive(false); }

                        node:status[report:in][report:out=>(parent;)]
                        {
                                // Check to see if all subordinates are done
                                // If not set transmission flag in out to false
                        } } } }
```

Figure 9-15 Unit.proc; Basic unit construct in the Brigade2 demonstration

```
{import process {unit} }
{import message {order, report} }
{
        process:soldier(unit)
        {
                mode:waiting_for_orders
                {
                        node:receive[order:ord][report:out=>(me;):(sub_times[0])]
                        {
                                // Set report time to some random point in the future.
                        } } } }
```

Figure 9-16 soldier.proc; Declaration of the process:*soldier* construct in the Brigade2 demo

9.2. Multiple Node Textual Simulations

The multi-node samples were developed to test the Time Warp mechanism for synchronizing nodes in a distributed simulation system. They are, like the single node examples in the previous section, rather simple, and serve primarily as a test of the underlying system.

9.2.1. Relay1

Relay1 is a simple reflector demo similar to the Ping demo described above. The main difference is that the two processes reside on different simulation engine instances. Figure 9-17 shows a simple schematic of the Relay1 demo. Figure 9-18 contains the relevant code segments.

Note that the **process:*relay*** construct is derived from the **process:*reflector***, so that the behavior of bouncing **message:*generic*** instances back to the sender is inherited in **process:*relay***. Also, in the declaration of the **process:*reflector*** instance in the **process:*relay*** construct, the affinity specification forces the subordinate reflector to be instantiated on simulation engine 1. The **process:*relay*** is the root process, and is therefore instantiated on simulation engine 0.

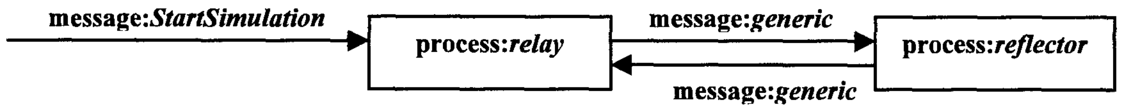


Figure 9-17 Message routing in the Relay1 demo

```
{import message {generic} }

{
        process:reflector
        {
                int:count(-1);              // Count of the number of messages received

...

                mode:Default
                {
                        node:reflect[generic:in][generic:out=>(in.getSource();)]
                        { count++; } }  } }
```

(a) reflector.proc

```
{import message {generic, StartSimulation} }
{import process {reflector} }

{
        process:relay(reflector)
        {
                reflector:r:1;              // Something to send a message to on simulation engine 1

                mode:Default
                {
                        node:start[StartSimulation:strt][generic:out=>(r;):(0.0)] { } } } }
```

(b) relay.proc

Figure 9-18 Source code for Relay1 process constructs

9.2.2. Relay2

Like the Relay1 demo, Relay2 has two processes that reflect messages between each other. The bootstrapping phase consists primarily of establishing a partnership between the two processes. This involves the root process, which is a **process:*relay*** construct, transmitting a message to its subordinate **process:*reflector*** instance that the two are partners (i.e. it informs the subordinate that it makes up the other half of the system). After this, the root process issues a message with time stamp 0.0 and sends it to the subordinate process instance. Upon receipt of a **message:*generic*** instance will return another **message:*generic*** instance to the sender. It will also send a **message:*generic*** instance to itself. Both of these messages have time stamps at some random time in the future. There is no termination condition in the program, so it theoretically can keep going forever.

What we end up with in this demonstration is a situation whereby for each message processed, two are generated. While this may be fine for abstract analysis, it will eventually cause problems with system memory as the number of pending messages grows linearly with the number already processed.

Figure 9-19 illustrates how the process instances interact. Figure 9-20 is the relevant code in the two process constructs.

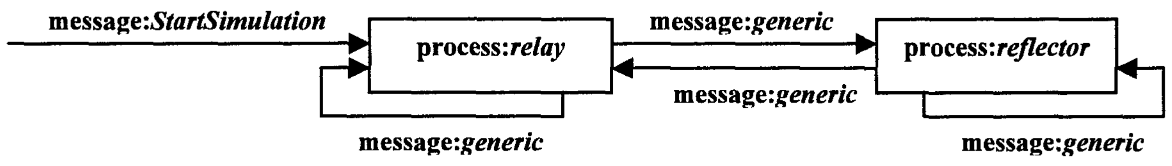


Figure 9-19 Relay2 demo message routing diagram

```
{import message {generic, set_partner, StartSimulation} }
{import process {reflector} }
{
        process:relay(reflector)
        {
                reflector:r:1;        // Something on another simulation engine to receive messages

                mode:Default
                {
                        node:start[StartSimulation:strt]
                                  [generic:out=>(r;):(0.0), set_partner:sp=>(r;):(-0.9)]
                                  { p[0]=me; p[1]=r; } } } }
```

(a) relay.proc

```
{import message {generic, set_partner} }
{
        process:reflector
        {
                process:p[2];                    // Pair of processes to send messages to
...
                mode:Default
                {
                        node:setPartner[set_partner:in][] { p[0]=me; p[1]=in.getSource(); }

                        node:reflect[generic:in]
                                [generic:out[2]=>(p[@];):(getTime()+random.nextDouble(10.0))]
                                { ... } } } }
```

(b) reflector.proc

Figure 9-20 Relevant code for the process:relay and process:reflector constructs

9.2.3. Relay3

```
{import message {StartSimulation, generic, setup} }
{import process {child} }
{
        process:relay
        {
                child:children[1000]:@%100+1;    // Distribute children evenly over 100 engines

                mode:Default
                {
                        node:start[StartSimulation:strt]
                                [setup:s=>(children;), generic:out=>(children;):(0.0)]
                        { s.set(children.size(), EngineStand::stand.engineCount()-1); } } } }
```

(a) relay.proc

```
{import message {generic, setup} }
{
        process:child
        {
                ulong:cc;                   // Number of children
                ulong:ec;                   // Number of engines

                mode:start
                {
                        node:relay[setup:in][]
                        {
                                // Setup cc and ed
                                start.setActive(false); } }

                mode:run
                {
                        node:relay[generic:in]
                                [generic:out:(getTime()+random.nextDouble(1.0))]
                        {
                                ulong di = random.nextInteger(cc);           // Destination index
                                out.addDest(process(di%(ec-1)+1, di/ec)); } } } }     // Set dest
```

(b) child.proc

Figure 9-21 Relevant code for the Relay3 sample

The Relay3 simulation has 1,001 individual processes distributed across 101 different engines. The only process on engine 0 is the root process, a **process:*relay*** instance. The remaining 1,000 processes are **process:*child*** instances evenly distributed across all of the remaining 100 engines. During initialization, the **process:*relay*** informs all **process:*child*** instances of some basic information regarding the number of processes and how they are distributed among the various engines. The simulation starts when the

process:*relay* instance sends a **message:*generic*** instance to all of the **process:*child*** instances. Each **process:*child*** will, in response to a **message:*generic*** input send another **message:*generic*** instance to a random **process:*child*** instance intended to be processed at some randomly determined time in the future. The simulation will run until the user terminates it. While statistically the number of pending messages should remain relatively constant, some engines may experience higher volumes of input messages than others, thereby unbalancing the system, possibly leading to excessive memory consumption,

The purpose of this sample was to test the roll back mechanism and to ensure that large numbers of engines were possible in terms of memory management issues. Figure 9-21 illustrates provides the relevant code segments for the simulation.

9.2.4. Relay4

The Relay4 sample uses a subscription process to allow broadcasts to multiple processes without regard to the actual membership of the subscription at the time the message is generated. This was initially used to test the SODL subscription feature, but was modified to use a user specified subscription process when the SODL subscriptions were removed.

There are seven processes in this sample. The only **process:*relay*** instance is the root process for the simulation. There are in addition two **process:*subscription*** instances and four **process:*child*** instances. Each **process:*child*** resides on a different simulation engine. During setup, each of the four **process:*child*** instances is informed of the process handles of the **process:*subscription*** instances. Each **process:*child*** then sends a **message:*subscribe*** to one of the **process:*subscription***, which will add that child to the subscriber list for that subscription.

Once completed, the **process:*relay*** instance will send a **message:*generic*** instance to both of the **process:*subscription*** instances. This message will be forwarded to all of the subscribers of each subscription, which should be each of the **process:*child*** instances.

```
import message {StartSimulation, generic, setup} }
{import process {child, subscription} }
{
        process:relay
        {
                subscription:sub[2];                // Subscription instances for the program
                child:children[4]:@;                // Child processes

                mode:Default
                {
                        node:start[StartSimulation:strt]
                                [setup:s=>(children;), generic:out=>(sub;):(0.0)]
                        { s.set(sub); } } } }
```

(a) relay.proc

```
{import message {generic, subscribe, unsubscribe} }
{import std {<set>} }

{
        process:subscription
        {
                std::set<process>:subscribers;      // The list of subscriber processes

                mode:Default
                {
                        node:subscribe[subscribe:in][] { subscribers.insert(in.getSource()); }
                        node:unsubscribe[unsubscribe:in][] { subscribers.erase(in.getSource()); }

                        node:forward[generic:in][generic:out[]]      // Forward a generic message
                        {
                                out.push_back(in);                   // Copy the input message
                                out.back().clearDest();              // Clear the destination list
                                std::set<process>::iterator i;       // subscribers iterator
                                for (i=subscribers.begin(); i!=subscribers.end(); ++i)
                                        out.back().addDest(*i);} } } }
```

(b) subscription.proc

Figure 9-22 Relay4 support process constructs

Upon receipt of a **message:*generic*** instance, a **process:*child*** will send three messages, each with randomly generated time stamps, and each to a randomly selected **process:*subscription*** instance. The first message is a **message:*subscribe***, the second a **message:*unsubscribe***, and the third a **message:*generic***. The **message:*unsubscribe*** will remove the sender from the subscription list for the receiving **process:*subscription*** instance. The other messages will exhibit the behavior described earlier.

The simulation will run until there are no longer any messages to process, or until the user stops the simulation. Relevant source code for the Relay4 sample is listed in Figures 9-22 and 9-23.

```
{import message {generic, setup, subscribe, unsubscribe} }
{
        process:child
        {
                ulong:ct;                       // Number of children
                long:di[3];                     // Destination index
                double:ts[3];                   // Outgoing message timestamp
                process:subscriptions[];        // Subscription process handles

                mode:start
                {
                        node:relay[setup:in]
                                [subscribe:sub=>(subscriptions[di[1]];):(ts[1])]
                        {
                                subscriptions = in.get();               // Get subscription handles
                                ct = ((ulong) subscriptions.size());    // Number of subscriptions
                                di[1] = random.nextInteger(ct);         // Join random subscription
                                ts[1] = -0.9;                           // Subscription event timestamp
                                start.setActive(false); } }             // Turn off the start mode

                mode:run
                {
                        node:relay[generic:in]                  // Generic inbound message
                                [generic:out=>(subscriptions[di[0]];):(ts[0]),
                                 subscribe:sub=>(subscriptions[di[1]];):(ts[1]),
                                 unsubscribe:unsub=>(subscriptions[di[2]];):(ts[2])]
                        {
                                count++;                                // Log the reflection
                                for (long i=0; i<3; ++i)                // Loop over the messages
                                {
                                        di[i] = random.nextInteger(ct);         // Get destination
                                        ts[i] = getTime()+random.nextDouble(1.0); } } } } }
```

Figure 9-23 process:*child* construct declaration for Relay4 sample

9.2.5. Relay5

The Relay5 sample was developed to debug a portion of the SODL run time system involving the Time Warp algorithm. There are three process instances, each on separate engines and all derived from the **process:*base*** construct. The root process is a **process:*source*** instance, which sends **message:*generic*** instances to itself and a **process:*relay*** instance it owns. In response, this **process:*relay*** then sends another **message:*generic*** to a **process:*sink*** that it owns. The **process:*sink*** does nothing except act as a message sink.

```
{import message {generic} }
{
        process:base
        {
                long:count(-1);                    // Counter
...
                mode:Default
                {
                        node:routine[generic:in][] { ++count; } } } }
```

Figure 9-24 process:base construct in the Relay5 sample

```
{import process {base} }
{process:sink(base);}
```

(a) sink.proc

```
{import message {generic} }
{import process {base, sink} }
{
        process:relay(base)
        {
                sink:s:me.getNode()+1;    // Sink for message stream instantiated of different engine

                mode:Default
                {
                        node:run[generic:in][generic:out=>(s;)] {} } } }
```

(b) relay.proc

```
{import message {StartSimulation, generic} }
{import process {relay, base} }
{
        process:source(base)
        {
                relay:r:me.getNode()+1;                // Relay process instantiated on a different engine
...
                mode:Default
                {
                        node:start[StartSimulation:s][generic:out=>(me; r;):(0.0)] {}
                        node:run[generic:in][generic:out=>(me; r;)] {} } } }
```

(c) source.proc

Figure 9-25 Code segments for instantiated Relay5 process constructs

The **process:*base*** construct, portions of which are listed in figure 9-24, provides support for a common output scheme and message counter for each of the derived constructs.

Relevant code from the **process:*base*** derived constructs is shown in Figure 9-25. Figure 9-26 illustrates how the messages are passed within the system.

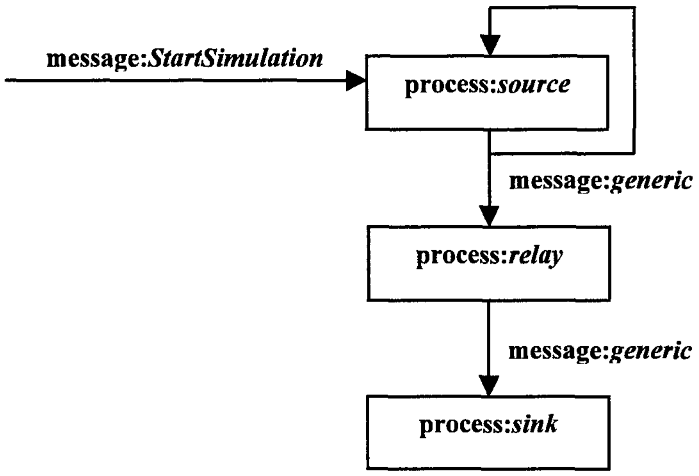


Figure 9-26 Message routine in the Relay5 sample

9.2.6. Relay6

On the surface, Relay6 may seem to be a relatively simple simulation, and the code itself would seem to confirm this. However, looking at a run of the simulation reveals some interesting dynamics that are not immediately obvious. The original intention of this sample was to test the basic functionality of the rollback mechanism to ensure that it was performing this task satisfactorily. It did this by running two processes at different "rates" (by which we mean the virtual time between successive messages delivered to each process was different) and then periodically sending a message from the slower process to the faster one, thereby inducing predictable rollbacks.

There are two processes, each on separate engines and derived from the **process:*base*** construct. The **process:*relay*** construct is the root process, and it sends a **message:*generic*** instance starting at time stamp 0.0. Upon receipt of a **message:*generic*** instance, the **process:*relay*** instance will send itself another at time *current_time*+0.1. Every tenth **message:*generic*** the **process:*relay*** receives will cause it to forward the next **message:*generic*** to both itself and a subordinate **process:*sink***. The **process:*sink***, on the other hand will, upon receipt of a **message:*generic*** issue another to itself with time stamp *current_time*+1.0. This message flow is depicted in Figure 9-27.

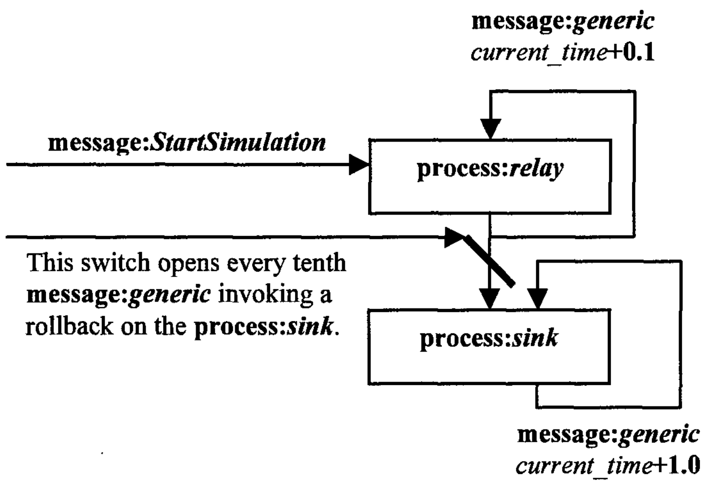

message:*generic*
current_time+0.1
message:*StartSimulation*
process:*relay*
This switch opens every tenth message:*generic* invoking a rollback on the process:*sink*.
process:*sink*
message:*generic*
current_time+1.0

Figure 9-27 Message routine in the Relay6 sample

```
{import message {generic} }

{
        process:base
        {
                long:count(-1);    // Counter
...
                method:round(public; double; double:t;) { return floor(t*10.0+0.5)/10.0; }

                mode:Default
                {
                        node:routine[generic:in][] { ++count;} } } }
```

Figure 9-28 Relay6 process construct for the process:*base*

The **process:*base*** construct, portions of which are shown in Figure 9-28, governs output and counting to provide some state information. It also declares **method:*round*** to eliminate minor differences in time stamps that can occur between the two derived process time stamps.

The interesting dynamics comes into play when the number pending **message:*generic*** instances for the **process:*sink*** instance begin to grow linearly with the number of messages it receives from the **process:*relay***. If not mitigated, there would be one pending message in the **process:*sink*** for every **message:*generic*** sent to it from the **process:*relay***.

This problem can be resolved by requiring the **process:*sink*** instance to send new **message:*generic*** instances to itself only when sufficient time has passed since the last transmission. This check is performed in the **node:*run*** declaration in **process:*sink***, which sets the transmission flag to **false** when the last message processed prior to the one currently under consideration has a time stamp that is too close to the current timestamp. Relevant code for the **process:*base*** derived classes is shown in Figure 9-29.

```
{import message {generic, StartSimulation} }
{import process {base, sink} }
{
        process:relay(base)
        {
                sink:s:1;           // Sink for the message stream
                ulong:ct(0);        // Counter

                mode:Default
                {
                        node:start[StartSimulation:in][generic:out=>(me;s;):(0.0)] { }

                        node:run[generic:in][generic:out=>(me;):(round(getTime()+0.1))]
                        { if (++ct%10==0) out.addDest(s); } } } }    // Send to s every 10 times
```

(a) relay.proc

```
{import message {generic} }
{import process {base} }
{
        process:sink(base)
        {
                double:lt(-1.0);    // Time stamp of last message:generic received

                mode:Default
                {
                        node:run[generic:in][generic:out=>(me;):(round(getTime()+1.0))]
                        {
                                out.setTX((getTime()-lt)>0.01);      // Stop rampant messaging
                                 lt=getTime(); } } } }
```

(b) sink.proc

Figure 9-29 Relevant Relay6 code segments

9.3. GLUT based demonstrations

The GLUT demonstrations are somewhat more complex than demonstrations discussed thus far. This complexity is largely the result of having to create and mange the scene graph on the SODL side.

One of the main problems with the basic GLUT View Manager (GVM) is that in order to display properly, the simulation engine must schedule screen updates. All of the SODL processes that have potential input are required prior to updating the display, to post updated information about their state to the GVM. This takes place in the form of messaging, and becomes quite time consuming for event relatively few objects on the screen. Some of the samples below augmented the basic GVM to reduce these periodic redraw cycles to only one message, instead of the hundreds or thousands that would have otherwise been necessary.

9.3.1. Bounce1

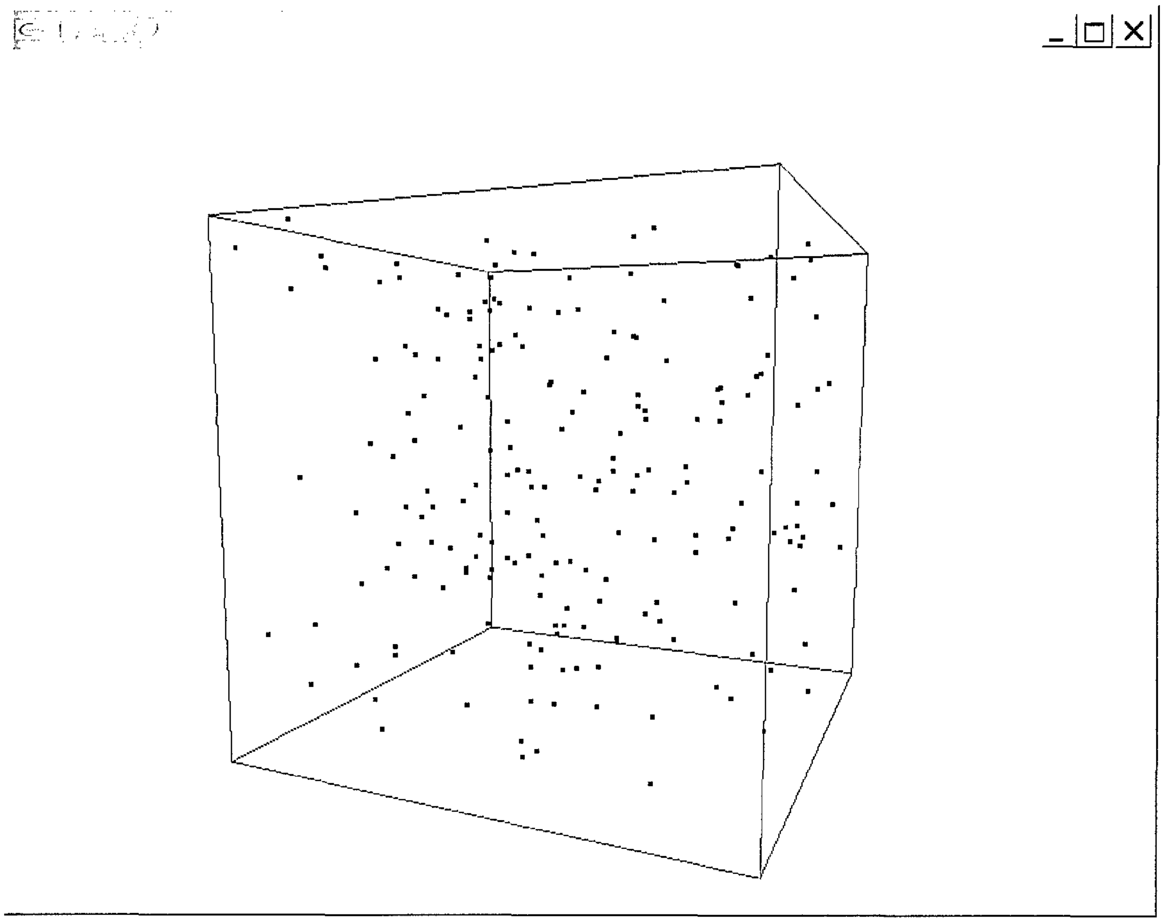


Figure 9-30 Output from Bounce1 demonstration

The Bounce1 demonstration depicts a cube with a collection of particles bouncing around the cube's interior. A sample output is provided in Figure 9-30. The simulation consists of a **process:*bounce*** instance.

```
{import message {hit, start, gr_update, SetVertex3D, AddVertex3D, SetSystem} }
{import process {Node3D, Vertex3D} }
{
        process:particle
        {
                Vertex3D:vrt;               // Screen vertex
                double:pos[3];              // Position vector
                double:vel[3];              // Velocity vector
                double:nextTime[3];         // Next impact times for each axis
                double:time(0.0);           // Time for the last velocity change

                method:init(public; void;) // Initialize particle position & velocity
                method:setNextHitTime(private; void; int:i;)// Get next hit time for axis i
                method:move(private; void;)         // Move particle to "current" position
                method:getMinAxis(private; int;)  // Return axis which will next have a collision

        mode:Default
        {
                node:start_sim[start:s] [hit:out=>(me;):(nextTime[out.axis])]
                { out.axis = getMinAxis(); }          // Schedule first collision

                node:update[gr_update:in][SetVertex3D:out=>(vrt;)]
                {
                        move();             // Move the particle to the current position
                        out.set(pos); }     // Update the vertex position

                node:change[hit:in][hit:out=>(me;):(nextTime[out.axis])]
                {
                        move();                             // Move particle to current position
                        vel[in.axis] = -vel[in.axis];       // Change particle the velocity
                        setNextHitTime(in.axis);            // Set next hit time for the specified axis
                        out.axis = getMinAxis(); } } } }    // Axis for the impact
```

Figure 9-31 ball.proc; Code governing ball motion and scene graph update

The user interface is a **process:*View3D*** instance, and includes a system defined **process:*Cube*** instance as well as a **process:*Polygon3D*** used to actually display the particles in the cube's interior. The simulation mainly concerns itself with the 200 particles bouncing around the interior of the cube. These particles, the behavior of which is declared in the **process:*particle*** construct, each have a **process:*Vertex3D*** that is added as a subordinate to the **process:*Polygon3D*** instance. As the simulation proceeds, the particles schedule the next "bounce" they have as a **message:*hit*** instance. The root process also schedules periodic updates of the scene, and broadcasts to all of the **process:*particle*** instances to report their position to the view so that the scene graph may be updated.

This leads to some problems with respect to performance, and points to areas where the SODL system has some limitations. In this instance, since each particle is responsible for explicitly updating its position in the view for each frame, there is a great deal of message traffic. Each **process:*particle*** instance, in each frame first must receive a request to update the scene graph, then forward an updated position its **process:*Vertex3D*** instance. From there, another message is forwarded to the actual view. The view then schedules an update to the scene graph in the form of yet another message. This final message queue has messages inserted in time stamp order, so no sorting is necessary. However, each of the other messages must be generated and processes in chronological order. This in turn needs to be done for each particle, in each frame, leading to a reduction in the overall performance. Relevant code for the **process:*particle*** is shown in Figure9-31 and message transmission is depicted in more detail in Figure 9-32.

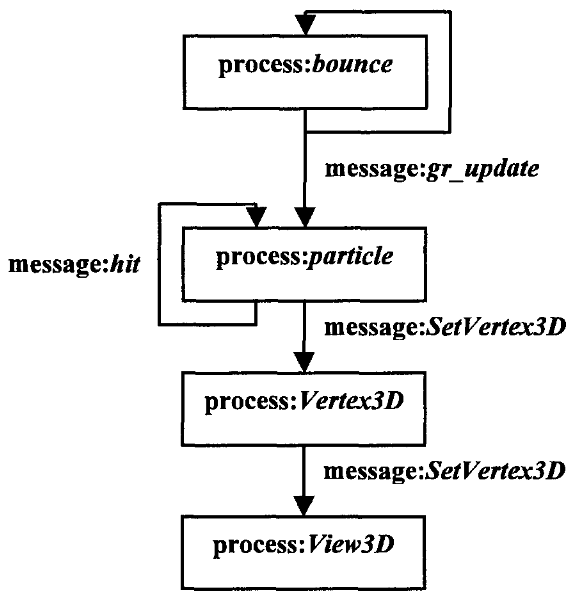


Figure 9-32 Messages for screen update in Bounce1 demo

9.3.2. Bounce2

The Bounce2 demo looks somewhat similar to the Bounce1 demo described above. It differs primarily in the load distribution between the SODL simulation engine and the GVM graphics engine.

The Bounce1 demo suffered from substantial performance degradation because each particle was required to provide the graphics engine with explicit position updates. This required an additional two messages per frame per particle.

There is in essence nothing changing between successive particle-wall collisions. The velocity of the particles remains unchanged between successive bounces, and a quick calculation based upon the location and time of the last bounce as well can accurately determine the location of the particle for anytime prior to the next bounce. The Bounce2 demonstration takes advantage of this by offloading these computations to an extension of the GVM. None of the scene update messages necessary in the Bounce1 demo are performed in Bounce2. The only events scheduled (after initialization) are for the particle-wall collisions, and periodic update requests to display the scene graph at the next time interval. The result is a significant improvement in performance.

Particle Count	***Bounce1 CPU time (seconds)***	***Bounce2 CPU time(seconds)***
1	2.202	1.876
5	1.938	1.765
10	1.766	1.906
50	0.704	1.876
100	1.375	1.876
500	10.94	1.890
1000	33.03	1.313
5000	Not Calculated	2.750

Table 9-1 Performance comparison of Bounce demos

The performance measurements displayed in Table 9-1 are taken from successive runs on a dual processor 700MHz Pentium III based computer system with an nVidia GeForce 256 based graphics card which provides hardware acceleration to perform geometric transformations. The tests were performed under the Windows 2000 SP1 operating system with the size of the window in which the simulation was displayed set to 800x600 pixels. The results shown are based upon 10 seconds of simulation time, amounting to about 400 rendered frames. The times listed do not include simulation initialization and process dependency setup. The image in Figure 9-33 shows the Bounce2 demonstration with 2,000 particles. Figure 9-34 shows the manner in which the messages are passed, and 9-35 the relevant code from the **process:*particle*** construct.

There were sufficient computing resources remaining to adjust the simulation somewhat and to incorporate gravity into the demonstration, as well. Thus, the particles in the final Bounce2 demonstration undergo parabolic motion. This feature was disabled during the test runs highlighted in Table 9-1.

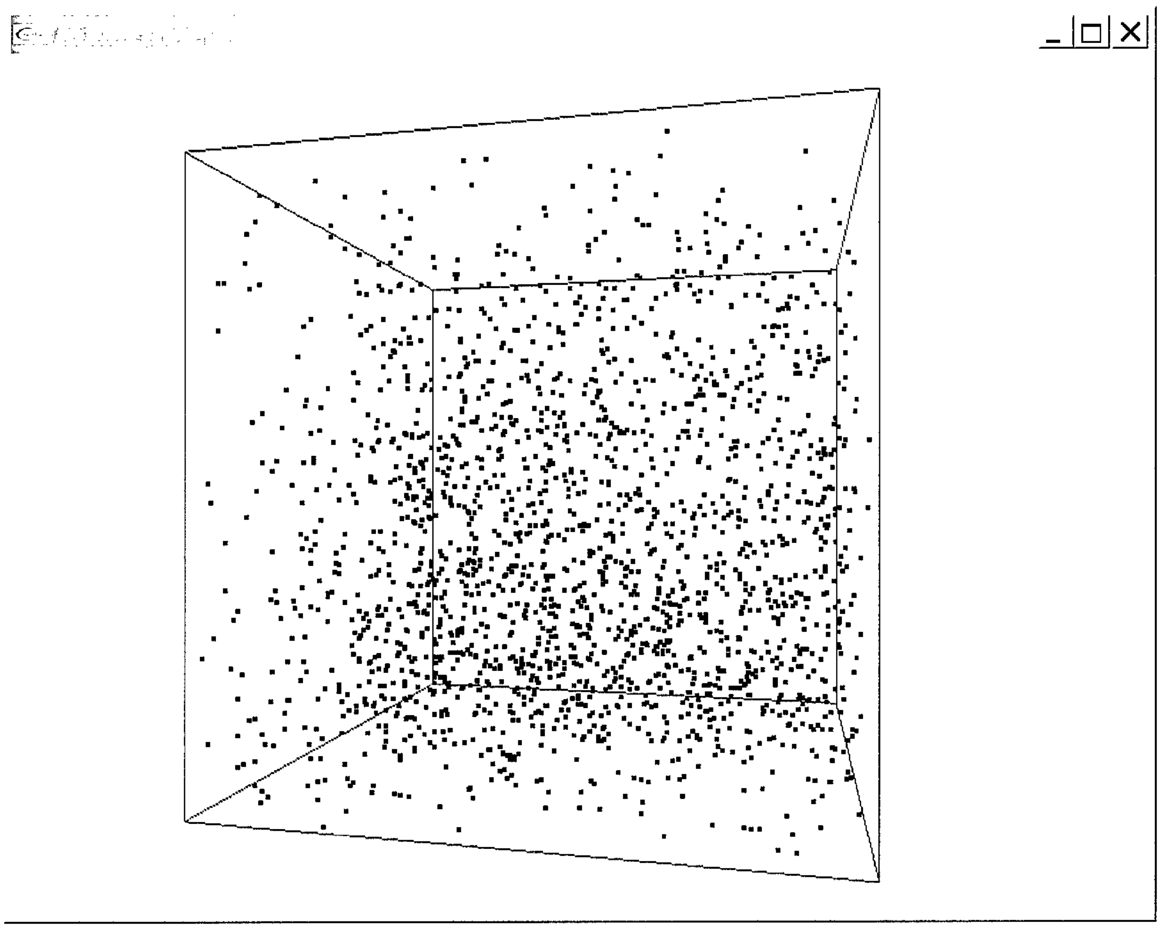


Figure 9-33 Bounce2 output with 2,000 particles

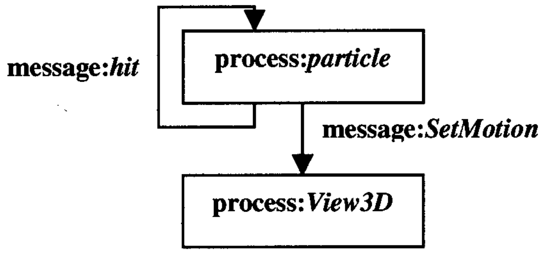


Figure 9-34 Messaging to update the scene graph in Bounce2

```
{import message {hit, set_motion, SetVertex3D, StartSimulation, AddVertex3D, AddView} }
{import process {Vertex3D} }
{import std {<vector>} }
{import {"Exception.h"} }

{
        process:particle(Vertex3D)
        {
                double:vel[3];              // Velocity vector
                double:acc[3];              // Acceleration vector
                double:nextTime[3];         // Next impact times for each axis
                double:time(0.0);           // Time for the last velocity change

                method:init(public; void;) { ... }   // Initialize the particle motion parameters
                method:setNextHitTime(private; void; int:i;) { ... }   // Get next hit time for axis i
                method:move(private; void;) { ... }           // Update particle state to current time
                method:getMinAxis(private; int;) { ... }      // Get the minimum time axis

                mode:Default
                {
                        node:start[StartSimulation:s][hit:out=>(me;):(nextTime[out.axis])]
                        { out.axis = getMinAxis(); }

                        node:addView[AddView:in][set_motion:out=>(in.getSource();)]
                        {
                                out.set(time,pos,vel,acc);          // Set parameters in new view
                                out.index = in.index; }             // Set the index value, too

                        node:change[hit:in][hit:out=>(me;):(nextTime[out.axis]),set_motion:sm[]]
                        {
                                move();                       // Move particle to current position
                                vel[in.axis] = -vel[in.axis];// Change the velocity
                                setNextHitTime(in.axis);  // Set next hit time for specified axis
                                out.axis = getMinAxis();  // Axis for the impact

                                std::map<process, gvm::object_index>::iterator i;       // For index
                                for(i=views.begin(); i!=views.end(); ++i)      // Loop over map
                                {
                                        sm.push_back(me);                     // Make new msg
                                        sm.back().addDest(i->first);          // Add destination to it
                                        sm.back().index = i->second;          // Specify the index
                                        sm.back().set(getTime(),pos,vel,acc); } } } } }
```

Figure 9-35 particle.proc – relevant code for updating scene graph in Bounce2

9.3.3. Brigade1

The Brigade1 demonstration is an extension of the Brigade2 demonstration described earlier. This extension provides a graphical representation of the state of progress during the brigade completing its task. A sample output is shown in Figure 9-36.

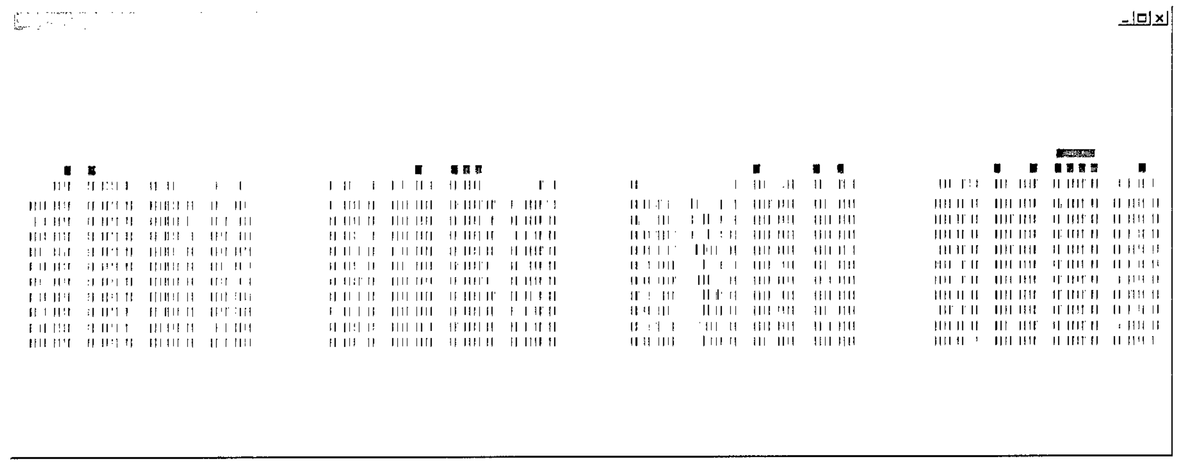


Figure 9-36 Brigade1 sample output

The top bar in the display represents the brigade, the four bars immediately beneath the four subordinate battalions, followed by the companies, platoons and squads. Under each squad is a column of ten soldiers. Red units are awaiting orders, yellow units are working on their assigned task, and green units have completed their task.

9.3.4. Hierarchy

The Hierarchy demonstration is a simple test of the two dimensional display capabilities of the GLUT view manager, and acts as a sort of predecessor to the Brigade1 demonstration above. It can be thought of as a binary tree that is being traversed. The scene starts out with all of the elements in the display colored blue. While a branch is being traversed, it changes to cyan. When the branch is finished being traversed, it changes color to green.

A sample output of the Hierarchy demonstration is depicted in Figure 9-37.

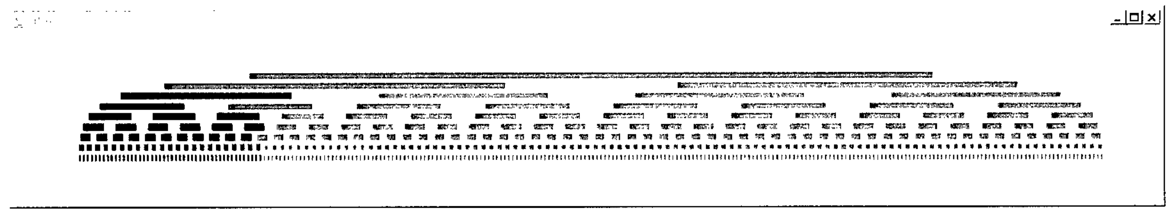


Figure 9-37 Output from the Hierarchy demonstration

9.3.5. Battle

This demo is the most complex of those discussed here. It is a hierarchical in its structure, with two opposing forces (designated RED and BLUE) consisting each of one company. Each company owns one command post, and five tank platoons. Each tank platoon has five tanks. The object is for one team to destroy the command post of the opposing force. There are three views associated with the demo; a 3D view of the environment and a tactical view for each force, representing the knowledge of the environment for the side related to the view.

The simulation starts with the two opposing forces in opposite corners of a square 400km^2 play field (20km to a side). All of the platoons of team RED initially move to take up a defensive position stretching across the center of the playfield. Two platoons of team BLUE move to defensive positions about 2,500m from the BLUE command post. The remaining three platoons move toward the RED command post.

9.3.5.1. Newtonian Motion

All of the objects in the simulation that are participants of the battle in one respect or another undergo Newtonian motion. This motion is broken into linear and angular components. Linear motion includes position within the play-space, linear velocity, and linear acceleration. Angular motion includes object orientation, rotation rate, and rotational acceleration. There is also a start and stop time associated with motion. The behavior directing the motion is incorporated into a C++ class, ***spt::NewtonianMotion***. This is done since the class is needed in multiple places within both the simulation engine and the rendering engine.

Changes in motion parameters are handled via message passing to **process:*NewtonianMotion*** instances. Each of these instances owns and manages the motion parameters in an ***spt::NewtonianMotion*** instance. Any changes to the motion parameters are passed to interested parties, namely the environment (see section 9.3.5.3) and any views rendering representations of the object undergoing Newtonian motion.

9.3.5.2. Sensing

Objects in the simulation have the ability to sense other nearby objects. Tanks have the ability to sense objects within a 2km range, while command posts can sense objects within a 5km range. To keep things simple, sensing objects are only notified when they detect enemy objects. Upon notification of a new sensor track, a tank will notify its parent platoon of the track. At present, the tank will then direct fire against the new track (see section 9.3.5.6 below). If the platoon did not previously know about the track, it will forward a notification to the parent company, and likewise await further orders. It will also maintain a list of tracks that subordinate tanks have sensed so that the platoon can make local decisions about how to deal with the situation. Upon notification to the company of the track, the company will update its list of tracks if it had not previously been aware of the track, and issue orders to address the situation. Figure 9-38 illustrates this situation.

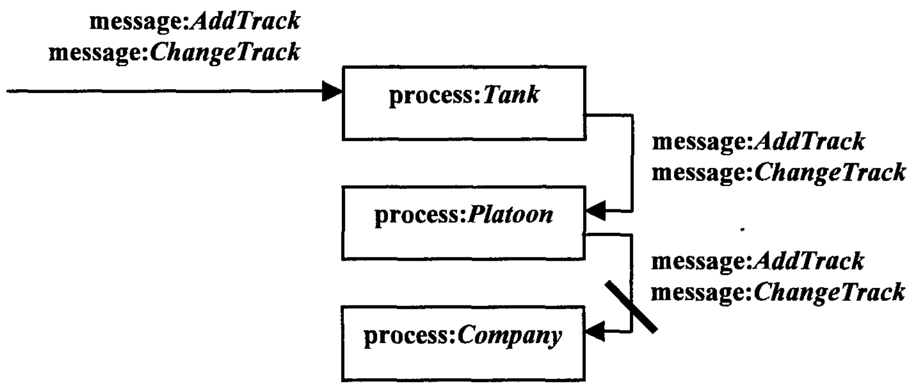


Figure 9-38 Sensor track detection and change notification in Battle demo

The procedure is the same for notification of changes in the track motion parameters. The switch between the platoon and company allows notifications to the platoon to be forwarded to the company only when such notifications from a subordinate tanks was new information. A similar notification mechanism is used for notifying parents of the loss of a sensor track. In that case, however, the platoon only forwards notification of the loss when all of its subordinate tanks had reported that it lost the track. This helps to reduce the number of messages that need to be passed from one level in the hierarchy to the next.

9.3.5.3. Environment

The environment is a process designed to govern interactions between objects in the simulation. It is primarily responsible for notifying objects of new tracks, changes in the track motion characteristics, and of lost tracks. The environment will also notify any objects affected by the impact of a munition that it was hit. Though the environment does not strictly perform collision detection between objects, it is logical to place it here in the event that this function is eventually incorporated into the simulation.

During initialization, each sensing or trackable object must register itself with the environment. This registration informs the environment as to the objects Newtonian motion parameters, sensing radius, and team membership (either BLUE or RED). An initial round of sensing event creation accompanies each new registration. In addition, any time an objects motion parameters change, the environment is notified so that a new collection of sensing events can be scheduled.

The environment schedules sensing events to occur at some point in the future. These scheduled events include detection events and loss events. Since there is no way to revoke sensing events that have already been scheduled, the environment performs one last check before the sensing object is informed to ensure that the sensor does in fact detect or lose the track, as appropriate. Only those sensing events that occur before both the track and sensor motion stop time will be scheduled. This also reduces the total number of messages that must be scheduled. Figure 9-39 illustrates this notification of track detection and loss.

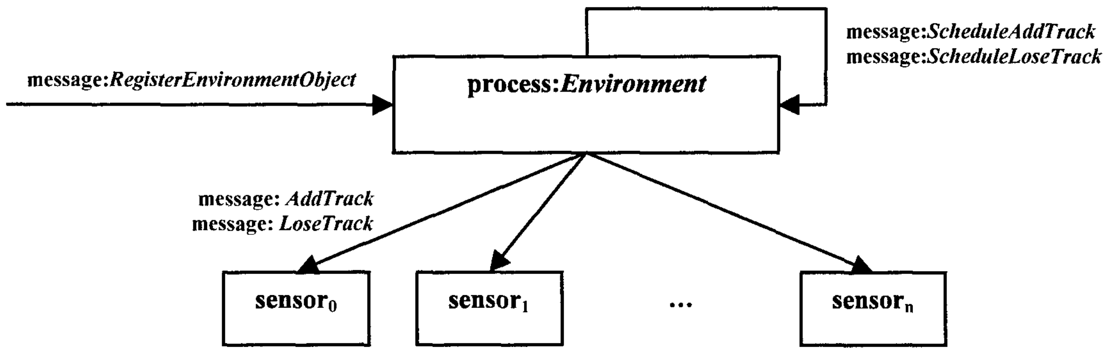


Figure 9-39 Notification of sensor track detection and loss

One assumption that is made regarding the sensor detection is that the sensors implicitly detect friendly units, but must explicitly detect enemy ones. This allows the environment to notify sensing objects only of detection events for enemy tracks, reducing the number of messages that must be passed any time an object changes it motion parameters.

9.3.5.4. Vehicle Movement

Tank motion is derived from the fact that it inherits this behavior from the **process:*Vehicle*** construct. Since there is really only one type of vehicle, encapsulating vehicle motion in this manner is, strictly speaking not necessary. It does provide a convenient mechanism for adding new vehicles to the simulation in the event that is eventually desired.

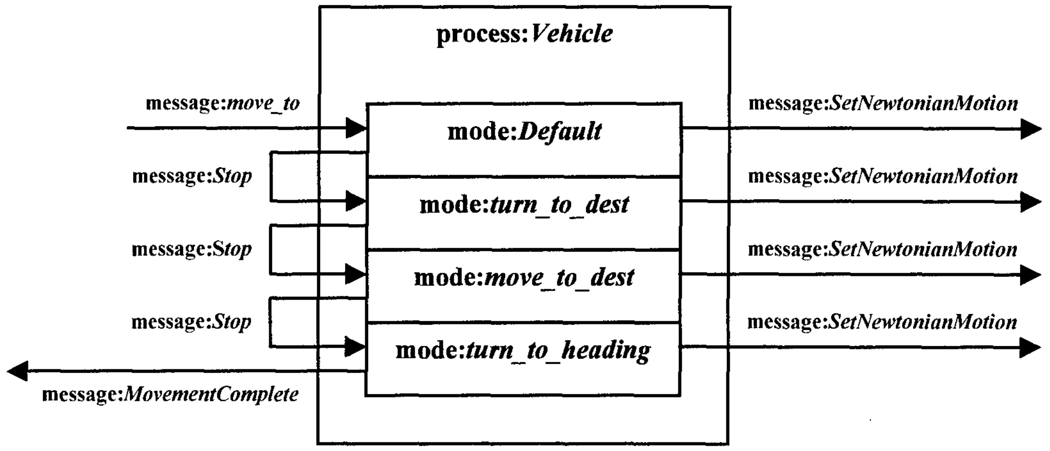


Figure 9-40 Messages governing vehicle motion in the Battle demo

Vehicles move about the battlefield when they receive movement request messages. These movement request messages contain a destination location and orientation. Upon receipt of one, the vehicle stops any motion that it may currently be performing and turns to face the destination. It then moves to the destination, and upon arriving there, will stop and turn to the final orientation. Each step in this sequence of events needs to be scheduled in the proper order. Furthermore, if a new movement request arrives during a move, the vehicle must disregard any commands issued in support of the initial move. To facilitate this sequencing of movement events, the vehicle switches modes as it prepares to perform the next phase of the movement command. Specifically, during the initial turn toward the destination, **mode:*turn_to_dest*** is active. Once the turn is complete, it is deactivated and **mode:*move_to_dest*** becomes active. Similarly, the final turn to the eventual heading is performed while **mode:*turn_to_heading*** is active. The transition from one phase of the movement to the next occurs when the vehicle receives a **message:*Stop*** instance it had previously scheduled for itself. At the beginning of each transition, **message:*SetNewtonianMotion*** messages inform the interested processes as to the new motion parameters for the vehicle. These processes are any GVM views with which the vehicle has been registered, the environment, and the parent object (in the case of tanks, the platoon to which the tank belongs). Upon completion of the final leg of the movement, the vehicle sends a **message:*MovementComplete*** message to its parent. Figure 9-40 illustrates this process.

9.3.5.5. Formation Movement

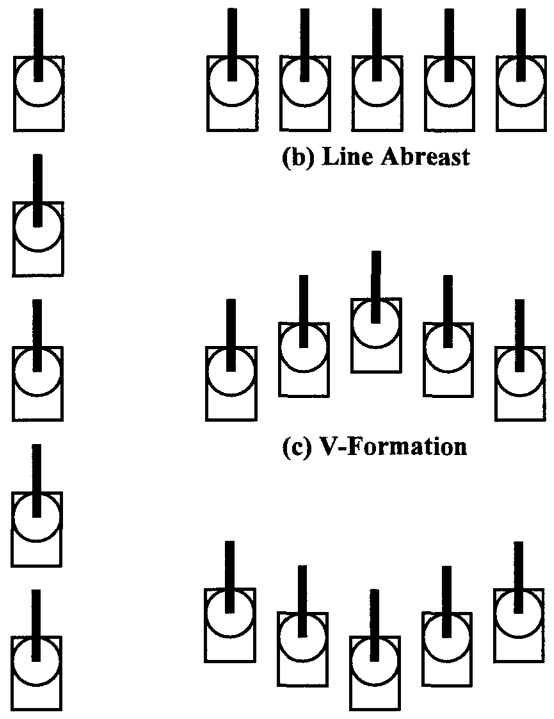


Figure 9-41 Predefined Tank Formations

Tanks are organized into platoons and are capable of moving in formations. These formations have a position, orientation, and left and right leg angles. These formations act as a template for positioning tanks relative to some reference point. Normally, the second (i.e. middle) tank acts as the lead for the remaining tanks. When ordered to a new position, the lead tank takes up that position, and the remaining tanks take up positions relative to the lead tank as specified by the formation structure. The orientation is a vector, the angle of which is used to dictate the direction the formation will face, and the magnitude the distance between adjacent tanks. The leg angles refer to a radial along which tanks will align themselves relative to the lead tank. A formation with a positive leg angle for a given side arranges tanks along that radial forward of the lead tank. Likewise, negative leg angles arrange tanks along a radial behind the lead tank. Figure 9-41 depicts some predefined formation arrangements. In this case, the column formation has a left leg angle of π/2 and a right leg angle of-π/2. The line-abreast formation has a left and right leg angle of zero. The V-Formation has a left and right leg angle of -π/8, and the forward sweep has left and right leg angles of π/8. Other formations are possible as well by explicitly specifying the left and right leg angles.

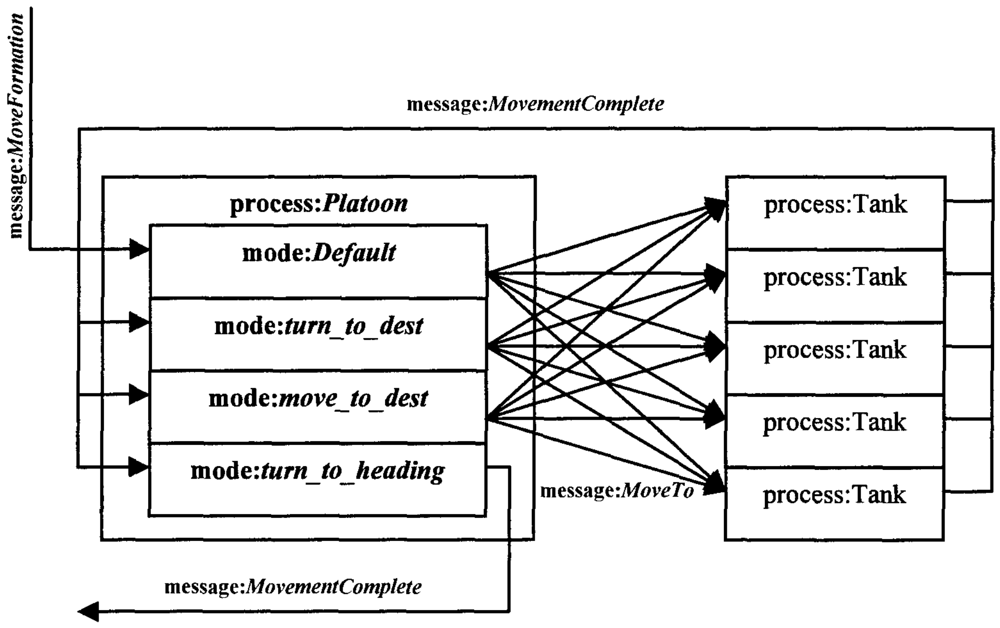


Figure 9-42 Messaging during formation movement

There are two message constructs used to establish a platoon formation. Both are derived from **message:*AdjustFormation***. The first, **message:*SetFormation*** is used during initialization to establish an initial formation. Use of this message will cause the platoon to explicitly specify the location and orientation of the tanks in the platoon. The second, **message:*MoveFormation*** is used to move a platoon from its current position, orientation and arrangement to some destination position, orientation and arrangement. When the platoon receives an instance of this second message type, it will turn the platoon to face the final destination, move the formation to the destination, and then face the platoon in the direction and arrangement specified by the destination orientation. Like vehicle movement, the platoon uses a collection of modes (**mode:*turn_to_dest***, **mode:*move_to_dest***, and **mode:*turn_to_heading***) to perform each phase of the movement. Each tank will report with a **message:*MovementComplete*** to the platoon that it has completed its movement instruction for that phase (as shown in Figure 9-40). Only when all tanks in the platoon are in the proper position will the next phase of the movement commence. The **process:*Platoon*** accomplishes this by issuing the next set of movement commands and transitioning to the next mode in the sequence. Figure 9-42 illustrates how the messages are passed from the platoon to the member tanks.

9.3.5.6. Fire control

Since the **process:*Environment*** only notifies **process:*Tank*** instances when they detect enemy units, any new track is assumed an enemy. This track is added to a target queue, and is associated with a **process:*SensorTrack*** instance in the simulation space. The **process:*SensorTrack*** construct is a parent construct for all **process:*Vehicle*** and **process:*CommandPost*** constructs. At present if the sensing tank has no other targets it is tracking, it will enter an attack mode whereby it will calculate a firing solution to the target and shoot a **process:*Munition*** at it.

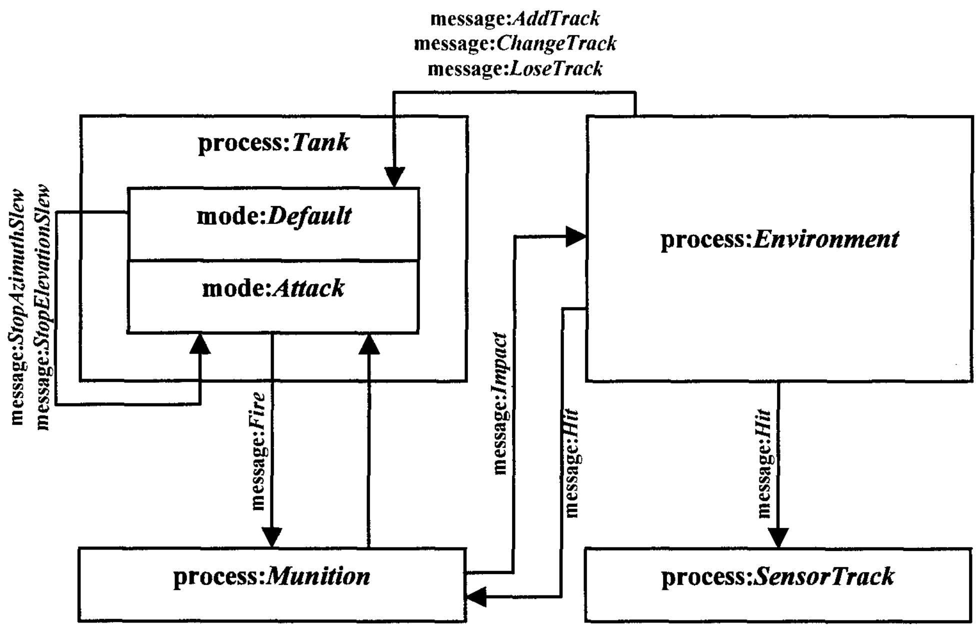


Figure 9-43 Fire control sequence for a tank

When firing at a target, the tank must first line up its gun so that when it fires the munition, it can be guaranteed of hitting the target. It does this by changing the azimuth and elevation of the gun, operations requiring some time to complete. While aiming, any changes in the target movement parameters cause the tank to recalculate the firing solution, and restart the aiming process. If the track is lost, it the tank will direct fire against the next target in its target queue. When the **process:*Tank*** finishes aiming its gun, it then fires a projectile. Upon impact, the **process:*Munition*** notifies the **process:*Environment*** of its impact location. The environment then looks at all objects registered to it and notifies any within 3m of the munition impact point that that it has been hit. The munition is also notified of the tracks it struck, so that it may inform the firing tank of the target(s) the munition destroyed. If the munition missed the intended target, the sequence is repeated until the target is destroyed. Once complete, the tank moves on to the next track that it has and repeats the process until all of its tracks are destroyed. Figure 9-43 illustrates the message passing in the fire control and subsequent notification.

9.3.5.7. Target Destruction

Once a target is struck with a munition, the simulation system assumes that it has been destroyed. There is some clean up that the system must perform afterwards to ensure that a destroyed object behaves that way. The first thing that must be done is for the destroyed object to inform its parent that it has been destroyed and to eliminate any tracks the parent may have as a result of the newly destroyed object's sensors. To this end, the object sends a **message:*Destroyed*** to its parent, followed by a collection of **message:*LoseTrack*** instances. In the case of a tank, the parent platoon also informs the **process:*Company*** instance so that the tactical view (see section 9.3.5.8) can be properly updated to reflect this loss. Likewise, if the object was the last in the parent unit to be tracking a particular enemy unit, then this must also be passed up the chain via another **message:*LoseTrack***. If the destruction of the subordinate unit results in the loss of the parent (i.e. all five tanks in a platoon are destroyed), then this must be passed further up the chain with another **message:*Destroyed*** instance.

The destroyed unit notifies the **process:*Environment*** that it has been destroyed[13]. Within the **process:*Environment***, several steps need to be taken to ensure that the objects sensing the newly destroyed unit can no longer track it. This is performed with one **message:*LoseTrack*** forwarded to all objects that had previously been able to sense the unit. The environment ignored events it may have scheduled based upon the future location of the now destroyed unit (a future sensor detection, for instance).

Figure 9-44 depicts some of the messages that need to be passed to perform this clean up.

[13] Even though the **process:*Environment*** notified the target of the fact that it was hit by a munition, it makes no assumptions about how many hits will actually destroy a target. This allows for some flexibility if the simulation is to one day be expanded.

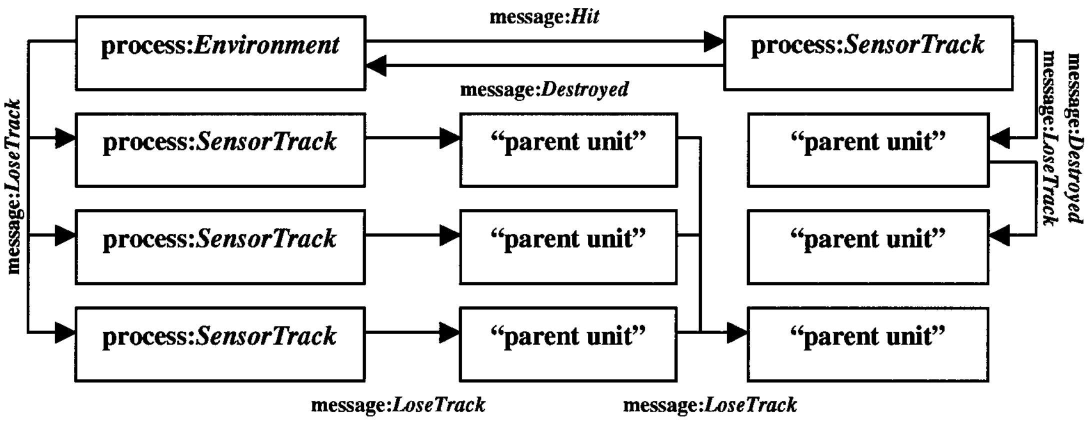


Figure 9-44 Clean-up after a unit destruction

9.3.5.8. Viewers

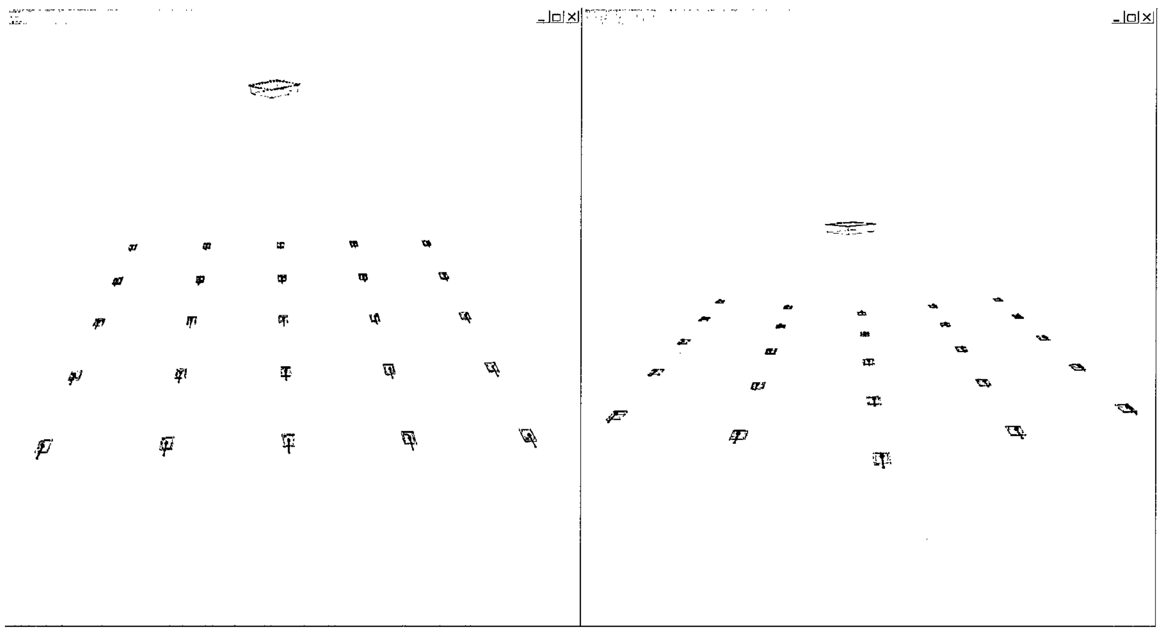


Figure 9-45 Shots of the initial platoon configurations in the BattleView of the Battle demo

Three viewers are used in the battle demo. The first is a three dimensional representation of the virtual environment. In this view, all of the objects can be seen with some degree of detail. Users can perform a virtual fly-by within this view to look at the layout of the various formations, or the location and orientation of an individual tank. Sample images are provided in figure 9-45.

The other views represent the world as seen by the Blue and Red teams individually. These are called the Red and Blue Tactical Views respectively. Friendly units in each view are colored with the team color, and have a disk around them that indicates the range of that particular unit's sensors. Hostile units will appear in their team color as they become visible to the friendly units. The tactical views for Figure 9-45 are provided in Figure 9-46.

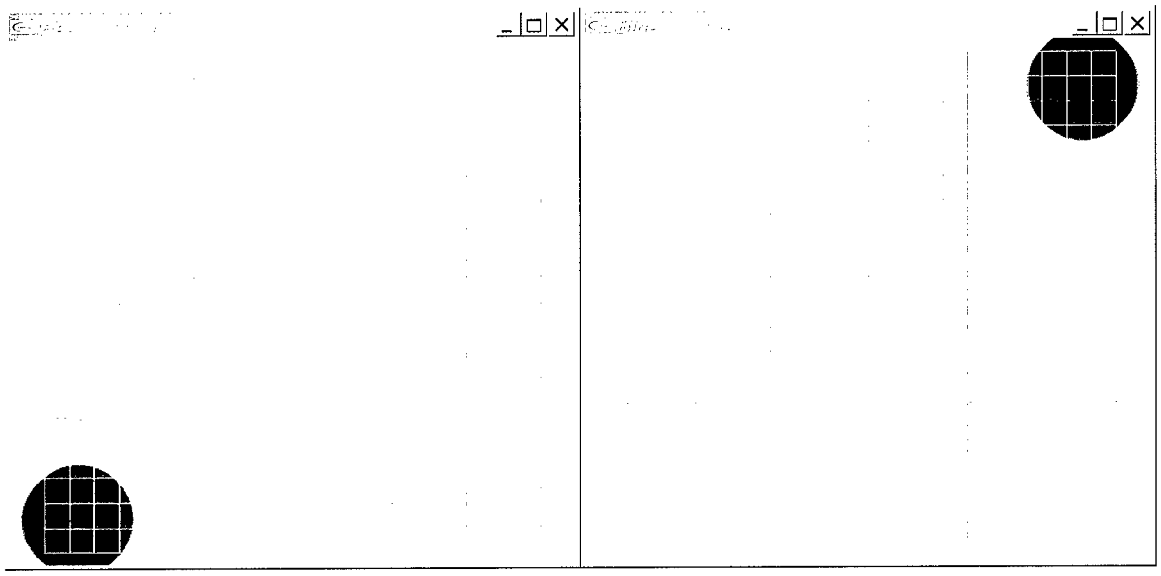


Figure 9-46 Red and Blue tactical views showing each side's knowledge of the environment

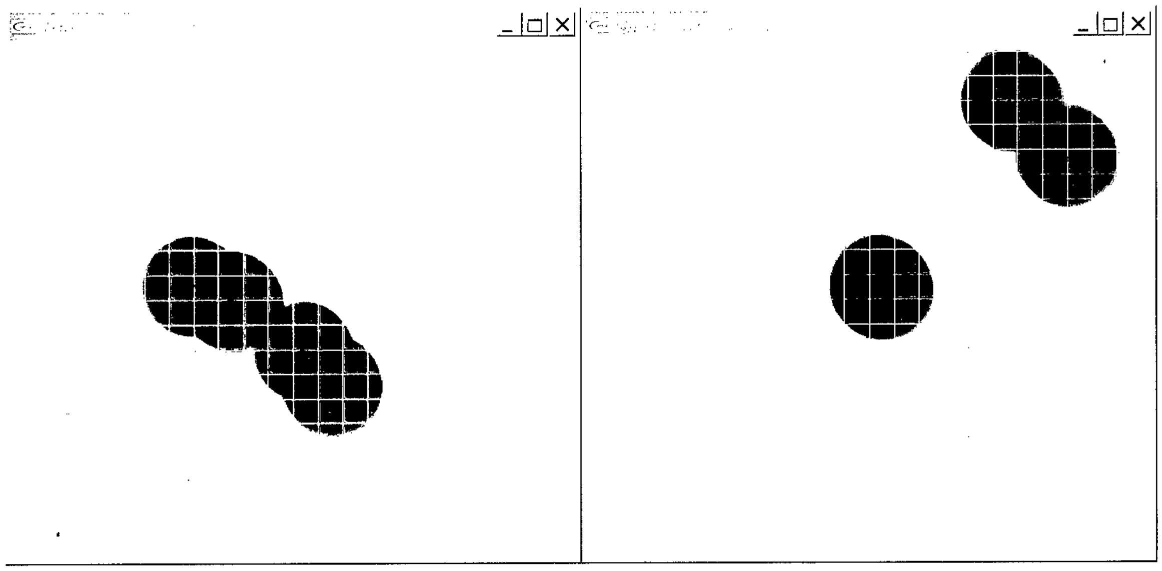


Figure 9-47 Tactical views shortly after the opposing forces encounter each other.

Figure 9-47 shows some additional tactical views as the simulation progresses, with its associated battle view depicted in figure 9-48.

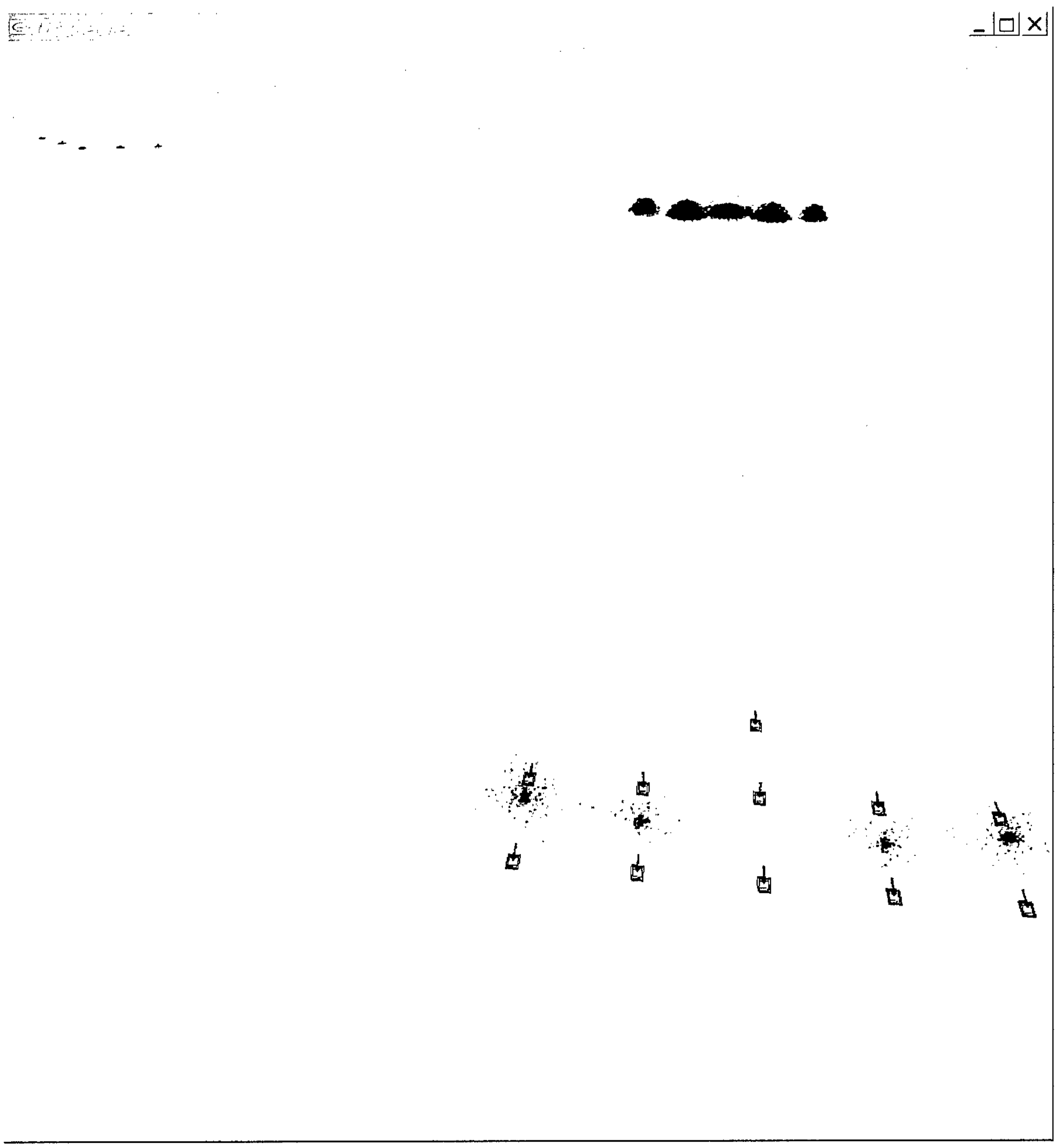


Figure 9-48 Sample engagement of opposing forces

The user can change viewing parameters within the views. Table 9-2 shows the key/mouse commands used to change the view and to do other simple tasks. Most of the keys and all of the mouse commands operate only in the BattleView. The keys that work in the Tactical View are ESC, 'h', 'H', 'r', and 'R'.

As a side note, the simulation is in a pause state at the beginning of the run, and must be resumed in order for the simulation to progress.

Key/Mouse command	*Function*
‘a’, ‘A’	Translate the view to the left
‘d’, ‘D’	Translate the view to the right
‘e’, ‘E’	Translate the view down
‘h’, ‘H’	Halt (pause) the simulation
‘q’, ‘Q’	Translate the view up
‘r’, ‘R’	Resume simulation
‘s’, ‘S’	Translate the view back
‘w’, ‘W’	Translate the view forward
1, 2, 3, 4, 5, 6, 7, 8, 9	Translation speed. n+1 translates at twice speed of n.
ESC	Quit the program
‘+’, ‘=’	Zooms into the scene
‘-‘, ‘_’	Zooms out of the scene
Mouse Left down & Drag left/right	Rotates the view about the view z axis
Mouse Left down & Drag up/down	Rotates the view about the view y axis
Mouse Right down & Drag left/right	Rotates the view about the view x axis
Mouse Center down & Drag up/down	Zooms into and away from scene

Table 9-2 BattleView keyboard/mouse commands

Chapter 10. Conclusions

Though we did not accomplish all we set out to with the SODL system, we have contributed to the body of knowledge, specifically in the field of distributed simulation.

10.1. Contributions of this work

10.1.1. SODL system

The SODL system is intended to provide a mechanism to facilitate development of distributed discrete event simulations based upon the notions of stimulus-response. Overall, the language structure successfully accomplishes this goal. The simulations highlighted in Chapter 9 and listed in Appendix C reveal that comparable systems developed from scratch would have had to contain considerable code to ensure that messages were delivered in the proper order. Likewise, systems built on top of existing libraries would have to include interfaces into those libraries that would again detract from actually defining the object behavior in the simulation system. Those fourth generation languages intended for use with either optimistic or conservative synchronization (namely YADDES and APOSTLE) require rigid specification of the message passing topology.

In the introduction, we claimed that the guiding principle of SODL was to split the simulation engine performing the mechanics of simulation from the behavior of the objects within the simulation. SODL largely succeeds at hiding many of the artifacts of performing a distributed simulation from the simulation system developer (the most notable exception being I/O operations) without sacrificing the generality available in other approaches. This allows developers to generate SODL code that closely resembles models they have developed without having the language or associated run-time system intrude upon that model. Additionally SODL provides an extensive (albeit non-exhaustive) collection of visualization tools to help facilitate analysis.

10.1.2. Simulation Formalism

Chapter 2 of this dissertation provides a formal description of the process of modeling and simulation and how it relates to real-world or hypothetical systems. This formalism provides a basis for discussing simulation within a larger context than has been provided within previous work. This formalism builds on top of the existing body of work and relates the larger context formally to the notions of distributed discrete event simulation prevalent in that existing work.

10.1.3. Asynchronous Global Virtual Time Algorithm

Chapter 3 concludes with the formal description of a generalized version of Mattern's algorithm for performing asynchronous Global Virtual Time (GVT) estimates. This generalization makes no assumptions about the underlying simulation topology in use for inter-process communications, while still maintaining the conditions necessary to ensure that local estimates of the GVT are lower bounds of the actual GVT. We go on to formally prove the correctness of this generalization. To the knowledge of this author, both the generalization and the formal proof of that generalization are original work.

10.2. Potential future work

While the amount of work that went into the SODL system is quite extensive, it falls short of some of the original notions surrounding it. This section highlights these issues, and introduces some others that are a natural extension of the work presented here.

10.2.1. Distributed SODL run-time system

The original intention of this work was to develop an operational distributed simulation system. While a number of factors seem to have played a role in keeping this feature out of the final system, ultimately it has been this author's responsibility for decisions made and actions taken that forced the decision to drop this capability.

The current SODL system implements a full optimistic simulation engine that can be fitted with the proper networking code to provide a fully distributed simulation system capability. The notion has always been to use the Message Passing Interface (MPI) as a means of distributing the simulation system, and it is this authors hope that this can be implemented in fairly short order.

10.2.2. Graphics Subsystem

While the GLUT View Manager (GVM) is useful as a research tool for visualizing the simulation system, more advanced approaches will likely require more sophisticated graphical representation, to include such things as solid rendering, lighting, curved surfaces, texturing, collision detection, and other features found in contemporary graphics systems. None of these advanced features are currently implemented in GVM. With some additional work, they could be incorporated easily.

10.2.3. User Interface

The user interface in the SODL system is very limited. Currently, it cannot be used to interact with objects in a rendered scene. There are a number of mechanisms within OpenGL allowing such interactions; such mechanisms could be used as a basis for allowing users to send messages to objects within the virtual environment. At the current time, the SODL system is not intended to support such efforts, and little thought has been given to how this might be accomplished.

10.2.4. Process Migration and Load Balancing

One problem associated with distributed simulation is load balancing, ensuring that no one node in a distributed simulation system has a significantly larger number of messages to process relative to its processor speed than other nodes. In conservative simulation, this problem leads to excessive blocking of the faster nodes, slowing down the overall simulation execution. In optimistic simulation, faster nodes will be required to use more memory to store old state and message information in the event a rollback is called for. This can be mitigated through process migration, directing a redistribution of the workload so that no one node is excessively burdened with a disproportionate workload.

This problem was never addressed in the SODL system, as it was beyond the scope of the research described herein, and because a distributed implementation of the SODL run-time system was never actually produced. If a distributed SODL run-time system is developed, an obvious mechanism to deal with these issues would be to actually continue instantiating all of the processes locally and then merely turning different processes on and off on different nodes depending upon load on each node.

10.2.5. Analysis tools

The focus of the SODL system has primarily been upon the language structure and to a lesser extent, the run-rime system. When simulation is used to perform analysis of one sort or another, it is quite often useful to provide tools to facilitate this analysis. This facilitation could be anything from formatting data that can be incorporated into an existing analysis tool, or through internal tools that can be called upon during or after a simulation run. There are no tools within the SODL system allowing a direct analysis of data generated. Such tools could be incorporated into later releases.

SODL also lacks any reasonable random number generation capability for serious analysis. (Press 1992) contains a number of algorithms for generating random numbers of various distributions. Such algorithms can be incorporated into a C++ class dedicated to random number generation. Alternatively, third party software with liberal copyright restrictions (e.g. GNU Public License) might also be useful.

10.2.6. Multiple inheritance

At times during the development of some of the demonstrations, the author discovered instances where multiple inheritance could be a useful tool. While work-arounds resolved many of the problems, they tended to be somewhat clumsy. As such, the overall system could greatly benefit from multiple inheritance.

Appendix A. SODL Language Parser Specification

The SODL Parser, sp, uses the following specification to parse SODL program files.

line-specifier : { *import-specifier* } *line-specifier*
| { **debug** *bool-value* } *line-specifier*
| { *message-specifier* }
| { *process-specifier* }

import-specifier : **import** *import-list*
| **import** *identifier* **::** *import-list*
| **import::** *import-list*
| **import message** *sim-import-list*
| **import process** *sim-import-list*

import-list : { *imports* }

imports : *C++-#include-parameter*
| *C++-#include-parameter* , *imports*

sim-import-list : *identifier*
| *identifier* , *sim-import-list*

message-specifier : **message :** *identifier* { *message-definition* }
| **message :** *identifier* ;
| **message :** *identifier* **(** *identifier* **)** { *message-definition* }
| **message :** *identifier* **(** *identifier* **)** ;

message-definition : *variable-specifier*
| *method-specifier*
| *variable-specifier message-definition*
| *method-specifier message-definition*

variable-specifier : *identifier* **:** *identifier process-qualifiers* ;
| *identifier* **::** *type-specifier* **:** *identifier variable-qualifiers* ;
| **::** *type-specifier* **:** *identifier variable-qualifiers* ;
| *scalar-specifier* **:** *identifier variable-qualifiers* ;

process-qualifiers : *null*
| *affinity-specifier*
| *size-specifier*
| *size-specifier affinity-specifier*

variable-qualifiers : *null*
| *initialization-specifier*
| *scalar-specifier*
| *size-specifier initialization-specifier*

type-specifier   *C++-type-expression*

affinity-specifier : **:** *modified-C++-integer-expression* ;

size-specifier : **[** *integer-value* **]**

initialization-specifier : **(** *modified-C++-expression* **)**

scalar-specifier : **bool**
| **byte**
| **char**
| **double**
| **float**
| **int**
| **long**
| **uint**
| **ulong**
| **rand**
| **process**
| **profile**

method-specifier : **method :** *identifier* **(** *method-parameter-list* **)** **{** *C++-code* **}**

method-parameter-list : *access-specifier* ; *type-specifier* ; *method-parameters*

access-specifier : **public**
| **protected**
| **private**

method-parameters : *null*
| *variable-specifier* ; *method-parameters*

process-specifier : **process :** *identifier* ;
| **process :** *identifier* **(** *identifier* **)** ;
| **process :** *identifier* { *process-definition* }
| **process :** *identifier* **(** identifier **)** { *process-definition* }

process-definition : *message-definition*
| *mode-declaration*
| *message-definition process-definition*
| *mode-declaration process-definition*

mode-declaration : **mode :** *identifier* **{** *node-list* **}**
| **mode :** *identifier* ;

Node-list : *node-specifier*
| *node-specifier node-list*

node-specifier : **node :** *identifier* **[** *input-message* **]** **[** *output-message-list* **]** **{** *C++-code* **}**

input-message : *identifier* **:** *identifier*

output-message-list : *null*
| *output-message* **,** *output-message-list*

output-message : *identifier* **:** *identifier output-qualifiers*
| *identifier* **:** *identifier* **[ ]** *output-qualifiers*
| *identifier* **:** *identifier* **[** *integer-value* **]** *output-qualifiers*

output-qualifiers : *null*
| **: (** *time-specifier* **)**
| **=> (** *destination-list* **)**
| **=> (** *destination-list* **) : (** *time-specifier* **)**

time-specifier : *modified-C++-double-expression*

destination-list : *modified-C++- destination* ;
| *modified-C++- destination* ; *destination-list*

identifier is an alpha-numeric string of characters starting with a letter. It can include the '_' character.

integer-value is C++ specification of an integer constant.

bool-value is one of the two constants, **true** or **false**.

C++-#include-parameter is a text stream that is suitable for inclusion immediately following an *#include* directive in a C++ source code file.

C++-type-expression is a C++ expression that describes a C++ type.

modified-C++-integer-expression is a C++ expression that when modified, will evaluate at run-time to an integer. It is modified by changing any instance of the '@' and '#' characters to an array index value and array size respectively.

modified-C++-expression is a C++ expression that when modified, will evaluate at run-time to a value of the desired type. It is modified by changing any instance of the '@' and '#' characters to an array index value and array size respectively.

C++-code is a block of C++ code.

modified-C++-double-expression is a C++ expression that when modified, will evaluate at run-time to a double precision floating point number. It is modified by changing any instance of the '@' and '#' characters to an array index value and array size respectively.

modified-C++-destination is a C++ expression that when modified, will evaluate at run-time to a process handle. It is modified by changing any instance of the '@' and '#' characters to an array index value and array size respectively.

Appendix B. SODL Run Time engine class reference

B.1. Overview

The SODL simulation engine and support library is designed to provide the basic infrastructure for passing messages between simulation processes. This documentation highlights the data members and methods for this infrastructure.

B.2. SODL Run-Time System C++ Classes

The SODL run-time system is responsible for ensuring that messages are delivered to the proper process in the proper time stamp order. It provides the basic infrastructure for this, and provides extensible class declarations for messages, processes, and support for IO operations. All classes in the Run-Time system are in the ***sodl::*** namespace unless otherwise stated.

B.2.1. ::Exception

The ::Exception class is a holding place for a collection of nested classes, each of which are different types of exceptions that the SODL run-time system may from time to time make use of when recognizing some problem from which it cannot recover. These nested classes are all publicly available and are described in the following sections.

Parent Classes: None

Derived Classes: None

B.2.2. ::Exception::BadCast

This exception class is used when an attempt to cast an object from one type to another (usually dynamically) fails. This is a somewhat unusual circumstance for the SODL system since the only objects that are normally cast from one type to another are derived either from ***sodl::Process*** or ***sodl::Message*** classes. Since these have fields for defining the actual type of the instance in question, dynamic casting should be straightforward. Thus, when an ***::Exception::BadCast*** is thrown, it is usually indicative of a deeper and more serious problem than simply a typing mix up.

Parent Classes: public *::Exception::Nonspecific*

Derived Classes: None

Protected Data Members:

std::string ::Exception::BadCast::from – String representation of the type being cast from.

std::string ::Exception::BadCast::to – String representation of the type being cast to.

Public Constructors:

::Exception::BadCast::BadCast(std::string** t, **std::string** f)* – This constructor initializes ***to to *t* and ***from*** to *f*. It also calls the parent constructor ***::Exception::Nonspecific***("Bad cast from").

::Exception::BadCast::BadCast(std::string** m, **std::string** t, **std::string** f)* – This constructor initializes ***to to *t* and ***from*** to *f*. It also calls the parent constructor ***::Exception::Nonspecific***(*m*).

Public Methods:

virtual void *::Exception::BadCast::seriailize(std::ostream&* *os*) const – This method displays the error message to stream *os*.

B.2.3. ::Exception::CausalityError

When a ***sodl::Engine*** instance receives a straggler with a time stamp *t*, and for some reason the engine or one of its subordinate process controllers cannot rollback to time *t* (due primarily to a programming bug) then the engine or process controller will throw an ***::Exception::CausalityError***.

Parent Classes: public *::Exception::Nonspecific*

Derived Classes: None

Protected Data Members:

double ***::Exception::CausalityError::att*** – Time stamp to which the SODL run-time system is attempting to rollback.

double ***::Exception::CausalityError::gvt*** – Last possible time to which this particular function can perform a rollback..

Public Constructors:

::Exception::CausalityError::CausalityError**(double** *g*, **double** *a*) – This constructor initializes ***gvt*** to *g* and ***att*** to *a*. It also calls the parent constructor ***::Exception::Nonspecific***("Causality error: Attempt at time ").

::Exception::CausalityError::CausalityError**(*****std::string*** *m*, **double** *g*, **double** *a*) – This constructor initializes the member variables ***gvt*** to *g* and ***att*** to *a*. It also calls the parent constructor ***::Exception::Nonspecific***(*m*).

Public Methods:

virtual void ***::Exception::CausalityError::seriailize*****(*****std::ostream&*** *os*) **const** – This method displays the error message to stream *os*.

B.2.4. ::Exception::Nonspecific

Any of a number of non-specific errors can be generated during the execution of a simulation instance. This exception is thrown when such an error is detected.

Parent Classes: None

Derived Classes: None

Protected Data Members:

std::string ***::Exception::Nonspecific::msg*** – Error message to display when called upon to do so.

Public Constructors:

::Exception::Nonspecific::Nonspecific*(*std::string* *m*) – This constructor initializes ***msg to *m*.

Public Methods:

virtual void *::Exception::Nonspecific::seriailize*(*std::ostream&* *os*) const – This method displays the error message to stream *os*.

B.2.5. ::Exception::RangeError

There are a number of ***std::vector*** instances throughout the SODL run-time system. When an attempt is made to access an element outside of the vector bounds, a range error is thrown.

Parent Classes: **public *::Exception::Nonspecific***

Derived Classes: None

Protected Data Members:

ulong *::Exception::RangeError::attVal* – Vector index that was attempted to be accessed.

ulong *::Exception::RangeError::size* – Actual size of the vector that is being accessed.

Public Constructors:

::Exception::RangeError::RangeError*(ulong** *s*, **ulong** *a*) – This constructor calls the parent constructor ***::Exception::Nonspecific("Range error: Attempt at index ") and initializes ***size*** to *s* and ***attVal*** to *a*.

::Exception::RangeError::RangeError*(*std::string* *m*, ulong** *s*, **ulong** *a*) – This constructor initializes the member variables ***size to *s* and ***attVal*** to *a*. It also calls the parent constructor ***::Exception::Nonspecific***(*m*).

Public Methods:

virtual void *::Exception::RangeError::seriailize*(*std::ostream&* *os*) const – This method displays the error message to stream *os*.

B.2.6. sodl::AntiMessage

sodl::AntiMessage instances are used in the Time Warp algorithm to revoke messages that have lost their validity in the simulation execution. The simulation engine (***sodl::Engine***) creates a ***sodl::AntiMessage*** when it becomes clear that messages transmitted need to be revoked. This is normally the result of a rollback to an earlier time than the current time stamp in the ***sodl::Engine*** instance issuing the ***sodl::AntiMessage*** instance.

Parent Classes: **public** ***sodl::SystemMessage***

Public Constructors:

AntiMessage::AntiMessage(***sodl::Message&*** *msg*) – Creates a ***sodl::AntiMessage*** instance that will, if sent to do so, revoke the ***sodl::Message*** instance *msg* and any copies made of it.

Public Methods:

static void ***AntiMessage::typeInit***(***sodl::*****mtype** *t*) – Performs type data initialization used in ascertaining the type of message instance transmitted.

virtual bool ***AntiMessage::annihilate***(**const** ***sodl::Message&*** *msg*) – Returns **true** if and only if this ***sodl::AntiMessage*** instance is supposed to annihilate *msg*.

B.2.7. sodl::Clock

The ***sodl::Clock*** class is responsible for managing time. It has a discrete mode where the ***sodl::Engine*** class can specifically set the clock time, and provides a framework for extending its functions to include operation in a real time fashion. Each ***sodl::Engine*** instance has one ***sodl::Clock*** instance. Processes controlled by a particular engine can call ***getEngine().getClock()*** to obtain a reference to their local clock.

Parent Classes: **public** ***sodl::TimeStamp***

Derived Classes: None.

Private Data Members:

static double ***sodl*::*Clock*::*pos*** – Used to determine the next possible time for current times strictly greater than 0. This is currently set to $1+10^{-15}$.

static double ***sodl*::*Clock*::*neg*** – Used to determine the next possible time for current times strictly less than 0. This is currently set to $1-10^{-15}$.

static double ***sodl*::*Clock*::*endTime*** – Used as delimiter as the last possible simulation time. No messages scheduled to occur after ***sodl*::*Clock*::*endTime*** will be handled. The default value for this member variable is 10^{307}.

static double ***sodl*::*Clock*::*startTime*** – The time stamp of the **message:*StartSimulation***. Its default setting is –1.

Public Constructors:

sodl*::*Clock*::*Clock*(ulong** *n*) – The primary purpose of this constructor is to call the parent constructor ***sodl*::*TimeStamp(-***sodl*::*Clock*::*endTime***, *n*) to establish the clock time stamp and associate it with a specific ***sodl*::*Engine*** instance.

Public Methods:

static double ***sodl*::*Clock*::*getEndTime*(void)** – Returns a copy of the static member variable ***endTime*** to the calling routine.

virtual double ***sodl*::*Clock*::*getNextTime*(void) const** – C++ does not provide a routine (of which this author is aware) that when given a double precision floating point number, *t*, will return the smallest double precision floating point number that is strictly greater than t. ***getNextTime***() fills this niche, albeit imperfectly. It will return *next-time*(*current-time*) as defined in Equation 6-1.

static double ***sodl*::*Clock*::*getStartTime*(void)** – Returns a copy of the static member variable ***startTime*** to the calling routine.

B.2.8. sodl::Defs

The ***sodl::Defs*** class is responsible for managing all of the system-defined types and some common routines of which various other classes in the SODL simulation run-time system can make use. Sp generates both *baseDir/buildSubdir/*Defs.h and *baseDir/buildSubdir/*Defs.cxx. This creates a different ***sodl::Defs*** definition for different programmer-defined simulation systems.

Parent Classes: **public** ***sodl::Trace***

Derived Classes: ***sodl::Clock***; ***sodl::Earlier***; ***sodl::Engine***; ***sodl::Handle***; ***sodl::IdleListener***; ***sodl::Later***; ***sodl::Message***, ***sodl::Process***; ***sodl::ProcessController***; ***sodl::ProcessMode***; ***sodl::ProfileTools***; ***sodl::Random***; ***sodl::Schedule***; ***sodl::ScheduleItem***; ***sodl::ViewManager***;

Public Enumerators:

enum ***sodl::Defs::MessageType*** – This is an enumeration of all of the message types. The enumerator names have the form ***SMT_message-type*** where *message-type* is the programmer defined type name of a message. The last enumerator in the list of them has name ***SMT_LAST***.

enum ***sodl::Defs::ProcessType*** – This is an enumeration of all of the process types. The enumerator names have the form ***SPT_process-type*** where *process-type* is the programmer defined type name of a process. The last enumerator in the list of them has name ***SPT_LAST***.

Private Data Members:

static ***std::vector<std::string> sodl::Defs::msgNames*** – A string representation of the message types. In general ***msgNames***[***sodl::Defs::SMT_****message-type*] = "*message-type*".

static ***std::vector<std::string> sodl::Defs::procNames*** – A string representation of the process types. In general ***procNames***[***sodl::Defs::SPT_****process-type*] = "*process-type*".

Protected Data Members:

static ***std*::*vector<std*::*vector*<bool> > *Defs*::*msgTypes*;** – Relationship between related message types. ***msgTypes***[*t1*][*t2*] (where *t1* and *t2* are both ***MessageType*** instances) is **true** exactly when *t1* is associated with a message class which is a parent of the message class associated with *t2* or *t1=t2*.

static ***std*::*vector<std*::*vector*<bool> > *Defs*::*procTypes*;** – Relationship between related process types. ***procTypes***[*t1*][*t2*] (where *t1* and *t2* are both ***ProcessType*** instances) is **true** exactly when *t1* is associated with a process class which is a parent of the process class associated with *t2* or *t1=t2*.

Public Methods:

static void *sodl*::*Defs*::*startup*(void) – Performs a number of static initialization functions including populating the static string arrays, ***msgNames*** and ***procNames*** as well as initialization of the ***msgTypes*** and ***procTypes*** arrays.

static void *sodl*::*Defs*::*shutdown*(void) – Performs functions associated with shutting down the simulation. As of this writing, this routine does not perform any specific function, but is provided as a counter-point to the ***startup***() method described above.

static bool *sodl*::*Defs*::*isType*(*sodl*::*Defs*::*MessageType a*, *sodl*::*Defs*::*MessageType b*) – A convenience function which returns ***msgTypes***[*a*][*b*] to the calling routine.

static bool *sodl*::*Defs*::*isType*(*sodl*::*Defs*::*ProcessType a*, *sodl*::*Defs*::*ProcessType b*) – A convenience function which returns ***procTypes***[*a*][*b*] to the calling routine.

static *std*::*string* *sodl*::*Defs*::*msgName*(*sodl*::*Defs*::*MessageType t*) – This returns the type name associated with ***MessageType*** *t*. Specifically, it returns the array value ***msgNames***[*t*] to the calling routine.

static *std*::*string* *sodl*::*Defs*::*procName*(*sodl*::*Defs*::*ProcessType t*) – This returns the type name associated with ***ProcessType*** *t*. Specifically, it returns the array value ***procNames***[*t*] to the calling routine.

virtual void *sodl*::*Defs*::*serialize*(*std*::*ostream*& *os*) const – A stub which may be used to produce class dependent formatted output to a stream. Any derived classes should overload it to properly format the output, to avoid the default output (" **** OVERLOAD ME **** "). It was not declared as an abstract method since there may be derived classes without any need to produce output.

B.2.9. sodl::Earlier

This class provides an operator for comparing the time stamps of pointers to two messages. It is used in the ***sodl::Engine*** class to properly order the messages in the event queue.

Parent Classes: public *sodl::Defs*.

Derived Classes: None

Public Methods:

static bool *sodl::Earlier::comp(sodl::Message* a, sodl::Message* b)* – This operator compares the time stamps on the two message pointers. It returns **true** exactly when the time stamp of **a* is earlier than the time stamp of **b*. In the event that the time stamp of the two messages are the same, the message handles are used, first comparing the engine index upon which the message was initially generated, and then the message instance number for that originating ***sodl::Engine*** instance.

virtual bool *sodl::Earlier::operator() (sodl::Message* a, sodl::Message* b)* – This merely returns the value returned by calling ***comp(a, b)***.

B.2.10. sodl::EndSimulation

The ***sodl::EndSimulation*** class is a message time stamped with the last possible value. Though its use may not be necessary, it made many aspects of the optimistic synchronization implementation employed in the SODL system somewhat more intuitive and straightforward.

Parent Classes: public *sodl::SystemMessage*

Derived Methods: None

Public Constructors:

***sodl*::*EndSimulation*::*EndSimulation*(void)** – This constructor initializes the message time stamp to ***sodl*::*Clock*::*getEndTime*()**, and calls the parent constructor in such a way as to make the root process the source of the message in all cases.

virtual bool *sodl*::*EndSimulation*::*getTX*(void) – Overloads ***sodl*::*Message*::*getTX*()** so that it always returns **true**.

static void *sodl*::*EndSimulation*::*typeInit*(*sodl*::mtype *t*) – Used to perform type initialization during the ***sodl*::*Defs*::*startup*()** call.

B.2.11. sodl::Engine

The ***sodl*::*Engine*** class is primarily responsible for message delivery for the processes it controls. It also manages the virtual time of all the processes under its control, directing fossil collection activities and rollbacks. It also manages the ***sodl*::*AntiMessage*** instances associated with messages that have been transmitted from subordinate processes.

Parent Classes: **public *sodl*::*Defs***.

Derived Classes: None

Private Data Members:

***std*::*priority_queue*<*sodl*::*AntiMessage**,*std*::*vector*<*sodl*::*AntiMessage**>,*sodl*::*Later*>**
sodl*::*Engine*::*antimessages – A list of pending ***sodl*::*AntiMessage*** instances. The top element of the event queue is compared with the top element of the antimessage queue. If they annihilate each other, they are both removed and destroyed and the original message is never processed.

sodl*::*Clock* *sodl*::*Engine*::*clock – The simulation time clock for the ***sodl*::*Engine*** instance.

std*::*priority_queue*<*sodl*::*Message**, *std*::*vector*<*sodl*::*Message**>, *sodl*::*Later*> *sodl*::*Engine*::*evQueue – List of pending messages, with the earliest message being at the top. Ties are broken using the message handle, so all they are executed in a unique order.

sodl::schedule sodl::Engine::fcSched – The fossil collection schedule. When new process states are saved for later rollback, they schedule a fossil collection event with the engine. This schedule is maintained in ***fcSched***.

bool ***sodl::Engine::hold*** – This contains **true** when this engine is in a hold status (i.e. it's waiting for the user to allow the simulation to proceed). It contains the value **false** otherwise.

ulong ***sodl::Engine::msgCount*** – The message count. Processes owned by a particular ***sodl::Engine*** instance will have as the index portion of their handle the current message count (***msgCount***). This value is then incremented in anticipation of the next message.

ulong ***sodl::Engine::node*** – This member acts as an index on the engine instance. Each ***sodl::Engine*** instance is given a unique node number to be used in the node portion of the handles for all processes the engine controls, as well as all messages originating from any such processes.

std::deque<sodl::AntiMessage> sodl::Engine::outMessages – ***sodl::AntiMessage*** instances associated with all messages transmitted from each engine are stored so that, when a rollback is necessary, the transmitted messages can be revoked. These ***sodl::AntiMessage*** instances are inserted into the double ended queue in the order they were created. Thus, they are sorted by generation time, making revocation and fossil collection a straightforward matter of removing elements from either the back or the front of the queue.

std::deque<sodl::Message*> sodl::Engine::processedMessages – Each message is processed in time stamp order. After being processed, they are inserted into the processed message queue, allowing rollbacks to occur should this be necessary. Since they are inserted into the front of this double-ended queue in time stamp order, rollback and fossil collection is simply a matter of removing from either front or back of the queue, respectively.

std::vector<sodl::ProcessController*> sodl::Engine::procList – This is the collection of process controllers governed by a ***sodl::Engine*** instance. As new processes are added, space in the vector is added to accommodate the ***sodl::ProcessController*** instances.

Public Constructors:

sodl*::*Engine*::*Engine*(ulong** *n*) – Initializes ***sodl*::*Engine*::*msgCount to 0, ***sodl*::*Engine*::*node*** to *n*, and calls the constructor ***clock*(*n*)**.

Public Methods:

virtual void *sodl*::*Engine*::*fossilCollect*(double *t*) – After incremental fossil collection is completed, regular fossil collection can be performed. This involves removing saved antimessages (from ***outMessages***) and previously processed messages (from ***processedMessages***) with time stamp values strictly earlier than *t*.

virtual *sodl*::*Clock*& *sodl*::*Engine*::*getClock*(void) – Returns a reference to the engine's clock ***clock***.

virtual *sodl*::*ScheduleItem* *sodl*::*Engine*::*getNextFossilCollectEvent*(void) – Returns to the calling routine a schedule item with the time stamp of the engine's next fossil collection event and index of the engine's node. Specifically, it returns to the calling routine ***sodl*::*ScheduleItem*(*time_stamp*, *node*)**, where time_stamp takes on the value ***Clock*::*getEndTime*()** if no fossil collection events remain, or to the time stamp of the next event ***fcSched*().*top*().*getTime*()** otherwise. This is used in the ***sodl*::*EngineStand*** to schedule engines for incremental fossil collections to ensure that all output and other irrevocable activities occur in the proper time stamp order.

virtual long *sodl*::*Engine*::*getNode*(void) – This method returns the node value to the calling routine.

virtual void *sodl*::*Engine*::*incrementalFossilCollect*(double *t*) – The top element in the fossil collection schedule, ***fcSched***, should have time stamp *t*, and is scheduled for process controller *n* on the engine. Process controller *n* is allowed to perform fossil collection up to time *t*, which should allow only the earliest fossil not previously collected to perform any irrevocable actions.

virtual bool *sodl*::*Engine*::*holding*(void) const – Returns true exactly when this engine instance is holding because of a scheduled or user induce hold in the simulation.

virtual void ***sodl::Engine::init*****(void)** – This routine performs some initialization for the ***sodl::Engine*** instance. This initialization includes calling the init method for each of the process controllers in ***procList***.

virtual ulong ***sodl::Engine::nextMessage*****(void)** – This routine returns the current value of the ***msgCount*** and then increments it by one for the next message. This value is used in ***sodl::Message*** instances to create a unique index for the message handle.

virtual ulong ***sodl::Engine::nextProcess*****(void)** – This routine allocates space in the process controller list (***procList***) and returns an index number to the calling routine. This is normally called by a process controller requesting space in the engine's process controller list for later registration.

virtual ***sodl::ProcessController&*** ***sodl::Engine::operator[]*****(ulong** *n***)** – Returns to the calling routine ***procList***[*n*], which is the n^{th} process controller on ***this sodl::Engine*** instance. If does not address a valid process instance on the local engine, an exception (***::Exception::RangeError***) is thrown.

virtual ulong ***sodl::Engine::processCount*****(void) const** – Returns the number of processes currently under control of the engine when called after the simulation starts. Prior to that time, there may be some process controllers that have not yet registered with the engine.

virtual void ***sodl::Engine::receive(sodl::Message&*** *msg***)** – Inserts an incoming ***sodl::Message*** pointer into ***evQueue***. If the time stamp on the incoming message is less than the current time in the ***sodl::Clock*** instance, then the engine instance rolls back to ensure that the incoming message can be in the correct order with respect to the other messages in the event queue.

virtual void ***sodl::Engine::reg(sodl::ProcessController&*** **pc)** – This registers a ***sodl::ProcessController*** instance with the simulation engine. This involves inserting the process controller into the proper location in the ***procList*** vector.

virtual void ***sodl::Engine::rollback*****(double** ***t*****)** – Conducts a rollback to time *t*. It does this by calling the rollback routines for each of the process controllers in the ***procList*** array; transmitting any members of ***outMessages*** with a time stamp not strictly less than *t*; removing all members of the double ended queue ***processedMessages*** with time stamps not strictly less than *t* and reinserting them into the ***evQueue***.

virtual void ***sodl::Engine::scheduleFC*(double** *t*, **ulong** *n*) – This schedules a fossil collection event in ***fcSched*** for process controller *n* at virtual time *t*.

virtual void ***sodl::Engine::scheduleFC*(*sodl::ScheduleItem*** *i*) – This routine schedules a fossil collection event at time ***i.getTime***() for process controller ***i.getIndex***().

virtual void ***sodl::Engine::serialize*(*std::ostream&*** *os*) **const** – This routine is designed to overloads ***sodl::Defs::serialize*(*std::ostream&*)** to produce output to stream *os* regarding various aspects of the engine state.

virtual void ***sodl::Engine::start*(void)** – This routine is called immediately prior to the simulation starting. It creates ***sodl::StartSimulation*** and ***sodl::EndSimulation*** messages and adds the processes under its control as destinations. These messages are given their default time stamp value (***sodl::Clock::getStartTime***()=-1 for ***sodl::StartSimulation*** instances, and ***sodl::Clock::getEndTime***() = 10^{307} for ***sodl::EndSimulation*** instances). These messages are then inserted into the pending message queue.

virtual double ***sodl::Engine::step*(void)** – This routine processes the next non-revoked message in the event queue provided its time stamp is not later than the earliest remaining hold in ***sodl::EngineList::stand.holdList***. It then returns the time stamp of the last message processed, or ***Clock::getEndTime***() if the event queue was empty.

virtual void ***sodl::Engine::transmit*(*sodl::Message&*** *msg*) – This routine retains a copy of *msg*'s antimessage for potential rollbacks. *Msg* is then sent to the transmit method of the ***sodl::EngineStand*** class governing local execution of the simulation system.

B.2.12. sodl::EngineStand

The ***sodl::EngineStand*** provides a mechanism for arbitrary distribution of ***sodl::Engine*** instances across a network. Each node in a distributed simulation, has exactly one ***sodl::EngineStand*** instance, and local copies of all the ***sodl::Engine*** instances that the programmer specifies. Each engine stand controls the activities of only those engines that reside on that stand's node. Messages destined for engines controlled by other engine stands must be forwarded over the network to simulation node controlling that engine.

Parent Classes: public *sodl::IdleListener*

Derived Classes: None

Protected Data Members:

bool *sodl::EngineStand::started* – This flag is set to **true** exactly when the simulation has started. It is **false** prior to that happening.

double *sodl::EngineStand::gvt* – Local estimate of the Global Virtual Time (GVT).

std::vector<sodl::Engine> sodl::EngineStand::engineList – This is a list of all of the engines in the simulation. Every ***sodl::Engine*** instance is controlled by exactly one ***sodl::EngineStand*** instance in the distributed simulation. However, all engines are allocated on all engine stands.

std::priority_queue*<double, *std::greater*<double> > *sodl::EngineStand::holdList – List of holds. If holdList is not empty, no engine may continue processing to times after the minimum element in the holdList. If the holdList is empty, then there are no holds, and processing may continue unabated.

Public Data Members:

static *sodl::EngineStand sodl::EngineStand::stand* – This is a static instance of the ***sodl::EngineStand*** class. Each node in a distributed simulation controls exactly one ***sodl::EngineStand*** instance. That instance is ***sodl::EngineStand::stand***.

sodl::ViewManager* sodl::EngineStand::vm – Each node in a distributed simulation has exactly one ***sodl::ViewManager*** instance associated with each ***sodl::EngineStand*** instance. That instance is ****stand::vm***.

Public Constructors:

sodl*::*EngineStand*::*EngineStand*(void)** – This constructor initializes member variables ***vm to ***NULL***, ***started*** to **false**, and ***gvt*** to the value returned from a call to ***sodl*::*Clock*::*getStartTime*()**.

Private Methods:

virtual void *sodl*::*EngineStand*::*updateGVT*(double *t*) – This method will update the local estimate of the GVT to time *t*. This update includes performing fossil collection on each of the engines the stand controls. The first phase of fossil collection involves polling each locally controlled engine as to the next scheduled item in their fossil collection schedule. Each engine is then scheduled at the engine stand level for incremental fossil collections; the engine with the earliest scheduled fossil collection event with time stamp less than t is permitted to perform that event. That engine is then polled for its new latest fossil collection event, which is then scheduled in the engine stand. All locally scheduled fossil collection events with time stamp less than *t* are thus performed in time stamp order.

The second phase of the fossil collection allows each of the locally controlled engines to perform its gross fossil collection (reclaiming memory occupied by processed messages and antimessages generated from outbound messages) up to time *t*.

virtual void *sodl*::*EngineStand*::*resize*(ulong *n*) – Resizes the number of engines to *n*. If *n* is larger than the current size of ***engineList***, then additional engines are allocated. Any requests to reduce the size of the engine list are ignored. During the process allocation phase of the simulation setup, there is no clear indication from the programmer exactly how many simulation engines will actually be needed. During the process allocation, as new engines are requested, they are dynamically added to the list of them in the engine stand. This method performs that resizing.

Public Methods:

virtual void *sodl*::*EngineStand*::*addHold*(double *t*) – This routine adds a hold at time ***t*** to the ***holdList***.

virtual ulong *sodl*::*EngineStand*::*engineCount*(void) – This routine returns the number of engines in the ***engineList*** vector.

virtual double ***sodl*****::*****EngineStand*****::*****getGlobalVT*****(void)** – Returns to the calling routine the value ***gvt***.

virtual ***sodl*****::*****ViewManager&*** ***sodl*****::*****EngineStand*****::*****getViewManager*****(void)** – This routine returns a reference to the view manager controlling the engine stand.

virtual bool ***sodl*****::*****EngineStand*****::*****holding*****(void) const** – Returns true exactly when all of engines controlled by this stand are holding due to the earliest remaining ***holdList*** item.

virtual bool ***sodl*****::*****EngineStand*****::*****idle*****(void)** – When the view manager has some idle time, it will call ***idle*****()**. When ****vm*** refers to a ***sodl*****::*****TextViewManager*** instance, the method is called until this method returns false (indicating that it has no more messages to process). When the controlling view manager is a ***sodl*****::*****GLUTViewManager*** instance, this method is called whenever the GLUT sub system has idle time.

virtual double ***sodl*****::*****EngineStand*****::*****nextHold*****(void)** – This routine returns the time of the next hold to the calling routine. If there are no holds pending, it returns ***sodl*****::*****Clock*****::*****getEndTime*****()*2.0.

virtual ***sodl*****::*****Engine&*** ***sodl*****::*****EngineStand*****::*****operator[]*****(ulong** *n*) – This method returns ***engineList***[*n*]. If the simulation has not yet started, and *n* is outside the range of the array, then the engine list is resized, prior to returning the specific engine instance. If the simulation has already started and *n* is outside of the range of the engine list, it throws an ***Exception*****::*****RangeError***.

virtual void ***sodl*****::*****EngineStand*****::*****setup*****(*****sodl*****::*****ViewManager&*** *v*) – This method just sets ***vm*** variable to &*v*.

virtual void ***sodl*****::*****EngineStand*****::*****start*****(void)** – This methods calls the start methods for each of the engines in the ***engineList*** vector. It also sets to **true** the ***start*** flag.

virtual void ***sodl*****::*****EngineStand*****::*****transmit*****(*****sodl*****::*****Message&*** *msg*) – This routine will forward a message to all engines with processes listed in the message destination list. If the engine is not locally controlled, the engine stand forwards it to the proper node in the distributed simulation.

B.2.13. sodl::GLUTViewManager

This class provides a framework allowing the GL Utility Toolkit (GLUT) to perform two and three-dimensional displays of simulation output and minimal support for allowing users to provide inputs to the simulation. Use of ***sodl::GLUTViewManager*** is specified by using the **–dglut** option in the SODL parser, **sp**. The ***sodl::GLUTViewManager*** makes use of the ***gvm::View*** class to perform actual IO operations.

Parent Classes: public *sodl::ViewManager*

Derived Classes: None

Protected Data Members:

std::vector<gvm::View*> sodl::GLUTViewManager::viewMap – The collection of GLUT windows managed by this ***sodl::GLUTViewManager*** instance. Each GLUT window is provided an index, and is referenced associatively with that index using the ***std::map*** template class.

static *sodl::GLUTViewManager* sodl::GLUTViewManager::manager* – This is the master view manager for simulation instances using GLUT for the system IO. This static instance is required because the callbacks from the various GLUT routines require calls to static methods. By retaining a static pointer, those static methods can access the view list to notify individual view instance of IO events.

Public Constructors:

sodl::GLUTViewManager::GLUTViewManager(sodl::IdleListener& *l*, **int*** *argc*, **char*[]** *argv*) – GLUT can process command line arguments to specify certain GLUT-specific aspects of the graphics interface. This constructor (which receives the values from the main program) passes the command line arguments here, and this method then passes them on a routine GLUT uses to parse input parameters, and pare out GLUT options specified therein. The parameters GLUT uses will be removed from the command line argument list before returning from the constructor. It will also invoke the parent class constructor by calling ***sodl::ViewManager::ViewManager***(*l*, *argc*, *argv*).

Public Methods:

virtual void ***sodl::GLUTViewManager::activateEntryListener*****(bool** *v*) – This starts listening for GLUT mouse window entry events when *v* is **true**, or deactivates listening for GLUT mouse window entry events when *v* if **false**.

virtual void ***sodl::GLUTViewManager::activateKeyboardListener*****(bool** *v*) – This starts listening for GLUT keyboard events when ***v*** is **true**, or deactivates listening for GLUT keyboard events when ***v*** if **false**. It activates or deactivates both the key press and key release callback listeners.

virtual void ***sodl::GLUTViewManager::activateMouseListener*****(bool** *v*) – This starts listening for GLUT mouse button events when *v* is **true**, or deactivates listening for GLUT mouse button events when *v* if **false**.

virtual void ***sodl::GLUTViewManager::activateMotionListener*****(bool** *v*) – This starts listening for GLUT active mouse motion events when *v* is **true**, or deactivates listening for GLUT active mouse motion events when *v* if **false**.

virtual void ***sodl::GLUTViewManager::activateOverlayListener*****(bool** *v*) – This starts listening for GLUT overlay events when *v* is **true**, or deactivates listening for GLUT overlay events when *v* if **false**.

virtual void ***sodl::GLUTViewManager::activatePassiveMotionListener*****(bool** *v*) – This starts listening for GLUT passive mouse motion events when *v* is **true**, or deactivates listening for GLUT passive mouse motion events when *v* if **false**.

virtual void ***sodl::GLUTViewManager::activateReshapeListener*****(bool** *v*) – This starts listening for GLUT reshape events when *v* is **true**, or deactivates listening for GLUT reshape events when *v* if **false**.

virtual void ***sodl::GLUTViewManager::activateSpecialListener*****(bool** *v*) – This starts listening for GLUT special keyboard events when *v* is **true**, or deactivates listening for GLUT special keyboard events when *v* if **false**.

virtual void ***sodl::GLUTViewManager::activateVisibilityListener*****(bool** *v*) – This starts listening for GLUT visibility events when *v* is **true**, or deactivates listening for GLUT visibility events when *v* if **false**.

virtual void ***sodl::GLUTViewManager::addView(gvm::View&*** *v*) – This sets ***viewMap***[*v.****getWindow***()] to &v. If necessary, the view map is resized to allow the new view to be added. The ***gvm::View*** instance is responsible for creating the GLUT window and retaining the proper value for that window's index.

static void ***sodl::GLUTViewManager::display*****(void)** – Callback function for handling GLUT display request events. It calls (****manager***)[***glutGetWindow***()].***display***().

static void ***sodl::GLUTViewManager::entry*****(int** *state*) – Callback function for handling GLUT window entry and exit events. State is the type of event (entry or exit). It calls (****manager***)[***glutGetWindow***()].***entry***(*state*).

static void ***sodl::GLUTViewManager::idle*****(void)** – Callback function for handling GLUT idle events, when GLUT is not busy handling other events. It makes a call to (****manager***).***idleListener.idle***() allowing the simulation to progress.

static void ***sodl::GLUTViewManager::keydown*****(unsigned char** *key*, **int** *x*, **int** *y*) – Callback function for handling GLUT key press events. The parameter *key* contains the key press value, and (*x*, *y*) is the screen position of the mouse at the time of the keyboard event. It calls (****manager***)[***glutGetWindow***()].***keydown***(*key*, *x*, *y*).

static void ***sodl::GLUTViewManager::keyup*****(unsigned char** *key*, **int** *x*, **int** *y*) – Callback function for handling GLUT key release events. The parameter *key* contains the value of the released key, and (*x*, *y*) is the screen position of the mouse at the time of the keyboard event. It calls (****manager***)[***glutGetWindow***()].***keyup***(*key*, *x*, *y*).

static void ***sodl::GLUTViewManager::motion*****(int** *x*, **int** *y*) – Callback function for handling GLUT active mouse motion events (i.e. with a mouse button depressed). (x, y) is the location of the mouse cursor. It calls (****manager***)[***glutGetWindow***()].***motion***(*x*, *y*).

static void ***sodl::GLUTViewManager::mouse*****(int** *button*, **int** *state*, **int** *x*, **int** *y*) – Callback function for handling GLUT mouse events. *button* is the button number that had the event, state is the button state, and (*x*, *y*) is the location of the mouse cursor. It calls (****manager***)[***glutGetWindow***()].***mouse***(*button*, *state*, *x*, *y*).

virtual ***gvm*::*View*& *sodl*::*GLUTViewManager*::*operator[]*(ulong** *n*) – Returns the ***gvm*::*View*** instance given by ***viewMap***[*n*].

static void ***sodl*::*GLUTViewManager*::*overlay*(void)** – Callback function for handling GLUT overlay events. It calls **(*manager)[*glutGetWindow*()].*overlay***().

static void ***sodl*::*GLUTViewManager*::*passive_motion*(int** *x*, **int** *y*) – Callback function for handling GLUT passive mouse motion events (i.e. with no mouse button pressed). (x, y) is the location of the mouse cursor at the time the event occurred. It calls **(*manager)[*glutGetWindow*()].passive_*motion***(*x*, *y*).

static void ***sodl*::*GLUTViewManager*::*reshape*(int** *width*, **int** *height*) – Callback function for handling GLUT window reshape events. The new width and height are given by the parameters *width* and *height* respectively. It notifies the ***gvm*::*View*** instance of the change by calling **(*manager)[*glutGetWindow*()].*mouse***(*width*, *height*).

static void ***sodl*::*GLUTViewManager*::*specialdown*(int** *key*, **int** *x*, **int** *y*) – Callback function for handling GLUT special key press events. *key* is the value of the key that was pressed, and (*x*, *y*) is the location of the mouse cursor at the time of the key press event. It calls **(*manager)[*glutGetWindow*()].*specialdown***(*key*, *x*, *y*).

static void ***sodl*::*GLUTViewManager*::*specialup*(int** *key*, **int** *x*, **int** *y*) – Callback function for handling GLUT special key release events. *key* is the value of the key that was released, and (*x*, *y*) is the location of the mouse cursor at the time of the key release event. It calls **(*manager)[*glutGetWindow*()].*specialup***(*key*, *x*, *y*).

virtual void ***sodl*::*GLUTViewManager*::*start*(void)** – Performs some initialization for starting up GLUT. This initialization involves establishing all of the listeners for various mouse, keyboard, and idle events. It then calls **::*glutMainLoop***() to start the simulation.

static void ***sodl*::*GLUTViewManager*::*visible*(int** *vis*) – Callback function for handling GLUT window visibility change events. It notifies the ***gvm*::*View*** instance of the change by calling **(*manager)[*glutGetWindow*()].*mouse***(*button*, *state*, *x*, *y*).

B.2.14. sodl::Handle

Handles are used in this system to reference and identify process and message instances. They are composed of two parts, a *node* and an *index*. Each process has a unique (in that no other process has the same) <*node*, *instance*> pair. Similarly, each message has a unique <*node*, *instance*> pair.

Parent Classes: **public** ***sodl*::*Defs***

Derived Classes: None.

Private Data Members:

ulong ***sodl*::*Handle*::*node*** – Node value for the handle.

ulong ***sodl*::*Handle*::*index*** – Index value for the handle.

Public Constructors:

sodl*::*Handle*::*Handle*(ulong *n*, ulong *i*)** – Initializes ***node and ***index*** to *n* and *i* respectively.

Protected Methods

virtual void *sodl*::*Handle*::*setNode*(long *n*) – Sets the ***node*** to *n*.

virtual void *sodl*::*Handle*::*setIndex*(long *i*) – Sets the ***index*** to *i*.

Public Methods:

virtual long *sodl*::*Handle*::*getNode*(void) const – Returns the ***node*** to the calling routine.

virtual long *sodl*::*Handle*::*getIndex*(void) const – Returns the ***index*** to the calling routine.

virtual bool *sodl*::*Handle*::*isType*(ptype *t*) const – Returns **true** if this ***sodl*::*Handle*** instance refers to ***sodl*::*Process*** instance of type *t*. This is accomplished by performing the call ***sodl*::*EngineStand*::*stand*[*node*][*index*].*isType*(*t*)**.

B.2.15. sodl::IdleListener

This is the base (abstract) class used by ***sodl::ViewManager*** to manage interactions between the user and the simulation engine.

Parent Classes: public *sodl::Defs*

Derived Classes: ***sodl::EngineStand***

Public Constructors:

sodl::IdleListener::IdleListener*(void)** – This is the default class constructor for the ***sodl::IdleListener class. It does not perform any special initialization.

Public Methods:

virtual void *sodl::IdleListener::start*(void)=0 – This abstract method is meant to be overloaded with simulation instance specific initialization routines.

virtual bool *sodl::IdleListener::idle*(void)=0 – This abstract method is meant to be overloaded with simulation instance specific instructions that run the simulation system through one iteration.

B.2.16. sodl::Later

This class provides an operator for comparing the time stamps of pointers to two messages. It is used in the ***sodl::Engine*** class to properly order the messages in the event queue.

Parent Classes: public *sodl::Defs.*

Derived Classes: None

Public Methods:

static bool *sodl::Later::comp*(*sodl::Message* a*, *sodl::Message* b*) – This operator compares the time stamps on the two message pointers. It returns **true** exactly when the time stamp of **a* is later than the time stamp of **b*. In the event that the time stamp of the two messages are the same, the message handles are used, first comparing the engine index upon which the message was initially generated, and then the message instance number for that originating ***sodl*::*Engine*** instance.

virtual bool ***sodl*::*Later*::*operator*()** **(*sodl*::*Message** *a*, *sodl*::*Message** *b*)** – This merely returns the value returned by calling ***comp*(*a*, *b*)**.

B.2.17. sodl::Message

This is the base class for all messages. Messages can contain data allowing information to be passed between process instances.

Parent Classes: public *sodl*::*Defs*, public *sodl*::*TimeStamp*

Derived Classes: ***sodl*::*SystemMessage*** and all user defined messages.

Private Data Members:

double *sodl*::*Message*::*genTime* – Time stamp of this message instance's creation.

Protected Data Members:

sodl*::*destination_list* *sodl*::*Message*::*dest – The destination list. Each destination process is listed, possibly more than once (in which case, the message will be delivered multiple times to the same process) in a compact form that allows rapid discovery of the destination engines and the processes on those engines listed as message recipients.

sodl*::*Handle* *sodl*::*Message*::*me – This is the message identifier. It is used to revoke messages when an inconsistent simulation state is encountered.

bool *sodl*::*Message*::*preempt* – This flag is normally **false**. If this flag is **true**, none of the default destination processes in the node header generating the message will be added after the node has completed the process state changes and message formatting. It will instead send only to the destinations in the destination list specified at the end of the node execution. The flag is normally set to **false** if the ***clearDest*()** method has been called somewhere in the node body.

process ***sodl*::*Message*::*source*** – This is the handle for the message's source process.

bool ***sodl*::*Message*::*timestampOverride*** – This is set to **true** if the time stamp value has been changed from inside the node body where the message originates. This is only of concern when a default message time stamp value is specified in the node declaration.

bool ***sodl*::*Message*::*tx*** – ***tx*** is an abbreviation for transmit. This is normally **true**. ***sodl*::*ProcessController*::*transmit*(*sodl*::*Message*&)** will check this value. If it is **true**, then the message will be forwarded to the intended recipients. If it is **false**, the message is discarded.

mtype ***sodl*::*Message*::*type*** – This contains the type information for the message instance.

Protected Constructors:

sodl*::*Message*::*Message*(long** *n*, **long** *i*, ***sodl*::mtype** *t*) – This constructor performs the initialization of the message by invoking the various constructors for the parent class and member variables. A call to the constructor ***sodl*::*TimeStamp*(-*sodl*::*Clock*::*getEndTime*(), *n*)** initializes the parent class. The call ***me*(*t*, *n*, *getEngine*().*getNextMessage*())** initializes the message handle. The message source is initialized by ***source*(*n*, *i*)**. ***genTime is initialized to the current simulation time. Flag values ***tx***, ***preempt***, and ***timestampOverride*** are initialized to **true**, **false** and **false** respectively.

sodl*::*Message*::*Message*(const process&** *p*, ***sodl*::*Handle *h*, **mtype** *ty*, **double** *t*)– This constructor is used for some ***sodl*::*SystemMessages***, notably the ***sodl*::*AntiMessage*** to set the various parameters of the message instance to the same as another message of some other type. Parent class initialization is performed through a call to the parent constructor, ***sodl*::*TimeStamp*(*t*, *p*.*getNode*())**. The message handle ***me*** is initialized with the copy constructor by setting it to *h* and ***type*** is set to *ty*. ***source***, the message source is also initialized with its copy constructor to *p*. ***genTime*** is initialized to the current simulation time. Flag values ***tx***, ***preempt***, and ***timestampOverride*** are initialized to **true**, **false** and **false** respectively.

Public Constructors:

Protected Methods:

virtual void ***sodl::Message::init*****(void)** – This method is intended to be overloaded by a simulation system developer to allow initialization of a message's content prior to being passed as a parameter to the process node that will eventually send the message. It does nothing in the ***sodl::Message*** class itself, however.

virtual ***sodl::destination_list& sodl::Message::getDest*****(void)** – This returns a reference to the message's destination list.

Public Methods:

virtual void ***sodl::Message::addDest*** **(process** *p***)** – Inserts the p into the destination list, ***dest***.

virtual void ***sodl::Message::addDest*** **(*****std::vector*****<process>** *p***)** – Inserts into the destination list, ***dest***, all of the **process** instances in *p*.

virtual void ***sodl::Message::clearDest*****(void)** – This clears the message destination list, ***sodl::destination_list::dest***, and sets the ***preempt*** to **true**.

virtual ***sodl::Message& sodl::Message::copy*** **(long** *n***)=0** – This abstract message is supposed to be overloaded by derived classes. It returns a copy of the message instance ****this*** and assigns its engine to ***sodl::EngineStand::stand***[*n*], which will manage it. This is used primarily when messages are transmitted from one ***sodl::Engine*** instance to another.

virtual ***sodl::Message& sodl::Message::copy*** **(void)=0** – This abstract method is intended to be overloaded by derived classes to returns a copy of ****this*** to the calling routine.

virtual double ***sodl::Message::getGenTime*****(void) const** – Returns the message generation time stamp, ***genTime***, to the calling routine.

virtual message ***sodl::Message::getID*****(void) const** – Returns ***me*** to the calling routine.

virtual ulong ***sodl::Message::getIndex*****(void) const** – Returns ***me.getIndex*()** to the calling routine.

virtual ulong ***sodl::Message::getNode*****(void) const** – Returns ***me.getNode*()** to the calling routine.

virtual process ***sodl*::*Message*::*getSource*(void)** – Returns ***source*** to the calling routine.

virtual bool ***sodl*::*Message*::*getTX*(void)** – Returns **true** exactly when the destination list is not empty and the ***tx*** flag is **true**.

virtual ***sodl*::mtype** ***sodl*::*Message*::*getType*(void) const** – Returns to the calling routine the value returned from calling ***me.getType*()**.

virtual bool ***sodl*::*Message*::*isPreempted*(void) const** – Returns to the calling routine the value in the ***preempt*** flag.

virtual bool ***sodl*::*Message*::*isType*(*sodl*::mtype** *t*) **const** – Returns **true** if and only if this message instance is of type or of sub-type *t*.

virtual void ***sodl*::*Message*::*serialize*(*std*::*ostream*&** *os*) **const** – Writes to the stream *os* a textual representation of the message instance.

virtual void ***sodl*::*Message*::*setTX*(bool** *TX*) – Sets the value ***tx*** to *TX*.

virtual *void* ***sodl*::*Message*::*setPreempted*(bool** *p*) – Sets ***preempt*** to *p*.

virtual void ***sodl*::*Message*::*setTime*(double** *t*) – Sets the value of the message time stamp override flag, ***timestampOverride***, to **true**, indicating that it should not be set to the default value in the node header declaration sending this message instance, and calls ***setTime*(*t*)**.

virtual bool ***sodl*::*Message*::*timeOverride*(void)** – Returns to the calling routine the value in ***TimestampOverride***.

static void ***sodl*::*Message*::*typeInit*(*sodl*::mtype** *t*) – Performs type data initialization used in ascertaining the type of message instance that is transmitted.

B.2.18. sodl::MessageHandle

This class serves as an identifier for message instances. It primarily provides a mechanism to distinguish between message handles and those for processes, since process handles provide a little more functionality.

Parent Classes: **public** ***sodl::Handle***.

Derived Classes: None

Public Constructors:

***sodl::MessageHandle::MessageHandle*(ulong *n*, ulong *i*)** – Calls ***sodl::Handle*(*n*, *i*)**.

B.2.19. sodl::Process

This is the parent class for all process constructs. It provides the basic functionality associated with all processes.

Parent Classes: **public** ***sodl::TimeStamp***, **public** ***sodl::Defs***

Derived Classes: All user-defined processes.

Private Data Members:

bool ***sodl::Process::collected*** – This is set to **true** when the ***sodl::Process*** instance has had its ***fossilCollect*()** method is called. This allows the instance to be retained after the initial fossil collection so that its state can be recovered in the event that it is needed in a rollback.

sodl::ProcessController* sodl::Process::controller – This is a pointer to the controller governing this process.

Protected Data Members:

process ***sodl::Process::me*** – This is a ***sodl::Handle*** instance with handle information about this process instance.

virtual void ***gvm::SetCubeSize::send*****(void)** – This methods actually sets the size attribute of the destination ***gvm::Cube*** instance.

B.4.23. gvm::SetCylinderSize

The ***gvm::SetCylinderSize*** message is intended to set the size attributes of a ***gvm::Cylinder*** instance.

Parent Classes: public ***gvm::Message***

Derived Classes: None

GLdouble ***gvm::SetCylinderSize::radius*** – Value to set the radius attribute of the destination ***gvm::Cylinder*** instance.

GLdouble ***gvm::SetCylinderSize::length*** – Value to set the length attribute of the destination ***gvm::Cylinder*** instance.

GLint ***gvm::SetCylinderSize::sides*** – Value to set the side count attribute of the destination ***gvm::Cylinder*** instance.

GLint ***gvm::SetCylinderSize::rings*** – Value to set the ring count attribute of the destination ***gvm::Cylinder*** instance.

Public Constructors:

gvm::SetCylinderSize::SetCylinderSize(gvm::View& ***v*****, double** ***t*****,** ***gvm::object_index i*****, GLdouble** ***ir*****, GLdouble** ***or*****, GLint** ***s*****, GLint** ***r*****)** – This constructor calls the parent class constructor ***gvm::Message(v, t,*** **GVM_SetCylinderSize,** ***i*****)** and initializes ***innerRadius***, ***outerRadius***, ***sides***, and ***rings*** to *ir*, *or*, *s*, and *r* respectively.

Public Method:

virtual void ***gvm::SetCylinderSize::send*****(void)** – This method commits the changes in the various attributes of the destination ***gvm::Cylinder*** instance.

static *sodl*::*Random sodl*::*Process*::*rand* – Random number generator for all ***sodl*::*Process*** instances.

sodl*::ptype *sodl*::*Process*::*type – Contains the type information for the specific process instance.

Protected Constructors:

sodl*::*Process*::*Process*(ulong *n*, ulong *i*, *sodl*::ptype *t*)** – This constructor calls the parent class constructor ***sod*::*TimeStamp*(-*Clock*::*getEndTime*(), *n*)**. It also calls the constructor for the process handle ***me*(*n*, *i*)**, sets ***type to *t* and ***collected*** to **false**.

Private Methods:

virtual void *sodl*::*Process*::*setCollected*(bool *c*) – Sets ***collected*** to *c*.

virtual bool *sodl*::*Process*::*isCollected*(void) – Returns ***collected*** to the calling routine.

Protected Methods:

virtual void *sodl*::*Process*::*instanceInit*(void) – The SODL parser, **sp**, will overload this function to perform instance specific initialization of various data members within the process definition. The simulation developer should not overload it.

Public Methods:

virtual void *sodl*::*Process*::*backup*(void) – During the state saving phase of the Time Warp algorithm, when process time stamps are increased, the old state is backed up and a new one is created. Once the new state is created with the new time stamp, the backup method for the new process is called. The programmer should overload this method in order to make use of this functionality and to manage aspects of the state saving that do not fall within the confines of the Time Warp algorithm.

virtual *sodl*::*Process*& *sodl*::*Process*::*copy*(void)=0 – This abstract method is intended to be overloaded by derived classes so that a copy of ***this*** can be returned to the calling routine. This is normally done for state savings purposes in the ***sodl*::*ProcessController*** instances.

virtual void ***sodl*::*Process*::*fossilCollect*(void)** – This routine is intended to be overloaded by the programmer. It is used to perform any irrevocable function required of the process prior to this particular process state being reclaimed.

virtual ***sodl*::*ProcessController&*** ***sodl*::*Process*::*getController*(void)** – Returns to the calling routine ***controller**.

virtual process ***sodl*::*Process*::*getID*(void)** – Returns ***sodl*::*Process*::*me*** to the calling routine.

virtual ***sodl*::ptype** ***sodl*::*Process*::*getType*(void)** – Returns to the calling routine ***type***.

virtual void ***sodl*::*Process*::*init*(void)** – Intended to be overloaded by programmer to perform application specific initialization.

static ulong ***sodl*::*Process*::*nextProcess*(ulong** *n*) – This is a convenience function returning the index of the next process to be added to engine *n*. It is accomplished by returning to the calling routine ***sodl*::*EngineStand*::*stand*[*n*].*nextProcess*()**.

virtual void ***sodl*::*Process*::*receiver*(*sodl*::*Message&*** *m*)**=0** – This is overloaded by the code generated by the SODL parser to process incoming messages. It will compare the actual message type of the reference *m* to the inputs expected by the nodes in active modes. A match occurs when a node accepts messages that are of the same type as *m* or a parent message type of *m*. When this occurs, the message is passed to the proper routine within the SODL parser generated code to handle the message. The process controller reference is passed since it is responsible for filtering messages and ensuring they are forwarded to the intended destinations.

virtual void ***sodl*::*Process*::*restore*(void)** – When a rollback occurs, a previous state is restored. That state for the process has its restore process called to perform any process specific rollback processing that might be required.

virtual void ***sodl*::*Process*::*serialize*(*std*::*ostream&*** *os*) **const** – This method allows a textual representation of the process state to be sent to the stream *os*.

static void ***sodl*****::*****Process*****::*****typeInit*****(*****sodl*****::ptype** *t*) – Initialize process type relations.

B.2.20. sodl::ProcessController

This class provides the basic functionality of the ***sodl*****::*****ProcessController*** instances. It is used to manage the flow of messages into an individual process.

Parent Classes: **public** ***sodl*****::*****Defs***

Derived Classes: None

Private Data Members:

std**::*****deque<sodl*****::*****Process*>*** ***sodl*****::*****ProcessController*****::*****stateQueue*** – This is the collection of SODL process states representing the state of the process at different points of time. Most recent states are at the back of the *std*::*deque*, and earlier ones are at the front. Fossil collection is done from the front, while any inbound messages are always delivered to the back element.

Public Constructors:

sodl**::*****ProcessController*****::*****ProcessController*****(*****sodl*****::*****Process*****&** *p*) – Calls constructor ***id*****(*****p*****.*****getID*****().*****getNode*****(),** ***p*****.*****getID*****().*****getIndex*****(),** ***p*****.*****getType*****())**. It then inserts *&p* into the back of the ***stateQueue*** and registers this instance with the controlling ***sodl*****::*****Engine*** instance.

Protected Methods:

virtual void ***sodl*****::*****ProcessController*****::*****backup*****(double** *t*) – This routine will back up the first state in the ***stateQueue*** by inserting a copy of it at the back of that data structure. That copy will have its time stamp set to time *t* and its backup method will then be called. This is performed in accordance with the state saving phase of the Time Warp algorithm.

virtual void ***sodl*****::*****ProcessController*****::*****displayStateQueue*****(*****std*****::*****ostream*****&** *os*) **const** – This is a routine which will send a formatted textual version of all the members of the ***stateQueue*** to stream *os*.

virtual void ***sodl::ProcessController::fossilCollect*****(double** *t*) – This routine performs incremental fossil collection. The controller first finds the ***sodl::Process*** instance with time stamp *t* in ***stateQueue***. If exists, this routine will call its ***sodl::Process::fossilCollect***() and set its ***sodl::Process::collected*** flag is set to indicate that it has been collected. Any elements of the state queue with time stamp values less than *t* are then removed from the queue and the memory they use is reclaimed. The element that was just collected remains until this ***sodl::ProcessController*** instance is tasked with another fossil collection.

virtual ***sodl::Engine&*** ***sodl::ProcessController::getEngine*****(void)** – This uses the process handle to retrieve a reference of the engine controlling this controller. The actual call is ***EngineStand::stand*****[*****id.getNode*****()]**.

virtual void ***sodl::ProcessController::rollback*****(double** *t*) – This routine causes a rollback to time *t* to occur for the process this ***sodl::ProcessController*** controls. This is accomplished by removing elements from the back of the controller's state queue, ***stateQueue***, until the back element had a time stamp value that is strictly less than *t*.

Public Methods:

virtual process ***sodl::ProcessController::getID*****(void)** – Returns to the calling routine ***stateQueue.back*****()-*****>getID*****()**.

virtual void ***sodl::ProcessController::init*****(void)** – Calls the init method for the back element in ***stateQueue***.

virtual void ***sodl::ProcessController::receive*****(*****sodl::Message&*** *msg*) – This routine will perform a backup of the back element of ***stateQueue*** if the time stamp of that element is strictly less than the time stamp of *msg*. It should not normally happen that the back element of the state queue has a timestamp which is strictly greater than that of *msg*, since the ***sodl::Engine*** that is now transmitting *msg* should have requested an engine-wide rollback to the proper time upon receipt of *msg*.

virtual void ***sodl::ProcessController::serialize*****(*****std::ostream&*** *os*) **const** – This routine provides a mechanism for providing formatted textual output to stream *os*.

virtual void ***sodl*::*ProcessController*::*transmit*(*sodl*::*Message*& *msg*)** – The back process in the state queue may elect, upon receipt of a message, to transmit new messages in response. That ***sodl*::*Process*** instance calls this method in that case, passing all outbound messages singly as the parameter. The first thing that is done is to ascertain whether to actually send the message. This is done by doing so only if *msg*.***getTX***() returns **true**. It also checks to see if the time stamp on the outgoing message is strictly greater than the current simulation time. If it is not, it could lead to an infinite loop and abnormal termination of the simulation run, so it is increased slightly in accordance with Equation 6-1. After this has been completed, the message is then forwarded to the controlling ***sodl*::*Engine*** instance for further processing.

B.2.21. sodl::ProcessHandle

This is primarily a minor extension of the sodl::Handle class. Though it does not contain any type information, it can be used to obtain type information about the process associated with this handle.

Parent Classes: **public** ***sodl*::*Handle***

Derived Classes: None

Public Constructors:

***sodl*::*ProcessHandle*::*ProcessHandle*(ulong *n*, ulong *i*)** – This constructor calls the parent constructor ***sodl*::*Handle*(*n*, *i*)**.

Public Methods:

virtual ***sodl*::*Defs*::ptype** ***sodl*::*ProcessHandle*::*getType*(void) const** – Returns to the calling routine ***type***. Referencing the type information available from the process controller associated with this process handle does this. The actual value returned is ***sodl*::*EngineStand*::*stand*[*node*][*index*].*getType*()**.

virtual bool ***sodl*::*ProcessHandle*::*isType*(*sodl*::ptype *t*) const** – This routine will return **true** if and only if the type associated with the process is type *t* or a sub-type of *t*. Referencing the type information in the process controller associated with this handle does this. The value returned is ***sodl*::*EngineStand*::*stand*[*node*][*index*].*isType*(*t*)**.

B.2.22. sodl::ProcessMode

Each mode declared in a SODL process becomes a ***sodl::ProcessMode*** instance in the associated C++ class. Prior to a message being delivered to a node, its parent mode must be polled for its activity level. This class provides the means for doing this.

Parent Classes: **public** ***sodl::Defs***, **public** ***sodl::TimeStamp***

Derived Classes: None

Private Data Members:

bool ***sodl::ProcessMode::active*** – Current state of the mode.

bool ***sodl::ProcessMode::newActive*** – State of the mode after the next time stamp change.

Public Constructors:

sodl::ProcessMode::ProcessMode**(ulong** ***n*****)** – This constructor initializes both of the member variables ***sodl::ProcessMode::active*** and ***sodl::ProcessMode::newActive*** to **true** and calls the parent constructor ***sodl::TimeStamp(-Clock::getEndTime(), n)***.

Public Methods:

virtual bool ***sodl::ProcessMode::isActive*****(void) const** – Returns to the calling routine ***sodl::ProcessMode::active***.

virtual void ***sodl::ProcessMode::serialize(std::ostream& os)*** **const** – Produces formatted textual output representing the state of this ***sodl::ProcessMode*** instance to stream os.

virtual void ***sodl::ProcessMode::setActive*****(bool** ***a*****)** – This will set the member variable ***newActive*** to ***a***. When the time stamp is updated, the this value will be copied to the ***active*** member variable.

virtual void ***sodl::ProcessMode::setTime*****(double** ***t*****)** – This will first make an explicit call to the parent class version, ***sodl::TimeStamp::setTime(t)***. It then copies the value in ***newActive*** to ***active***.

B.2.23. sodl::ProfileTools

This class is useful for timing the duration of various activities. It provides a processor-time clock to measures elapsed processor time since a reset or instantiation.

Parent Classes: public ***sodl::Defs***

Derived Classes: None

Protected Data Members:

::clock_t ***sodl::ProfileTools::time*** – Time of instantiation or last reset.

Public Constructors:

sodl::ProfileTools::ProfileTools**(void)** – Calls ***sodl::ProfileTools::resetTime*****()**.

Public Methods:

virtual void ***sodl::ProfileTools::resetTime*****(void)** – Sets the ***time*** to the current time using ***::clock*****()**.

virtual double ***sodl::ProfileTools::elapsedTime*****(void)** – Returns the elapsed processor time (in seconds) since ***time*** was last set (i.e. ***::clock*****()–*****time*****)**.

B.2.24. sodl::Random

This uses the standard C random number generator (***::rand*****()** and ***::srand*****()** for seeding) allowing users to uniformly distributed pseudo-random numbers. It is not a particularly divers random number generator, and end-users should probably consider replacing it with one that has a more varied and robust set of features.

sodl::Random has a typedef to ***sodl*****::rand**.

Parent Classes: public ***sodl::Defs***

Derived Classes: None

Public Constructors:

***sodl*::*Random*::*Random*(void)** – Initializes the random number stream by calling ***::srand*((unsigned) *::time*(*NULL*))**.

explicit *sodl*::*Random*::*Random*(uint *seed***)** – Initializes the random number stream by calling ***::srand***(*seed*).

Public Methods:

double *sodl*::*Random*::*nextDouble*(double *a***)** – Returns a uniformly distributed double precision floating point random number in the range [0, *a*).

double *sodl*::*Random*::*nextDouble*(double *a***, double** *b***)** – Returns a uniformly distributed double precision floating point random number in the range [*a*, *b*).

int *sodl*::*Random*::*nextInteger*(int *a***)** – Returns a uniformly distributed random integer in the range [0, *a*).

int *sodl*::*Random*::*nextInteger*(int *a***, int** *b***)** – Returns a uniformly distributed random integer in the range [*a*, *b*).

B.2.25. sodl::ScheduleItem

The ***sodl*::*ScheduleItem*** class is used for scheduling fossil collection events, though it could be used for other purposes as well. It is based on the ***std*::*pair*<double, ulong>** class, where ***first*** represents a time stamp, and ***second*** is an index of a process or engine that is scheduled.

Parent Classes: **public *std*::*pair*<double, ulong>,public *sodl*::*Defs***

Derived classes: None

Public Constructors:

sodl*::*ScheduleItem*::*ScheduleItem*(double** *t, ulong** *i***)** – This constructor calls the parent constructor ***std*::*pair*<double, ulong>** (*t*, *i*).

Public Methods:

virtual double ***sodl::ScheduleItem::getTime*(void) const** – Returns to the calling routine ***std::pair*<double, ulong>*.first***.

virtual ulong ***sodl::ScheduleItem::getIndex*(void) const** – Returns to the calling routine ***std::pair*<double, ulong>*.second***.

virtual void ***sodl::ScheduleItem::serialize*(*std::ostream*&** *os*) **const** – Produces formatted textual output of the ***sodl::ScheduleItem*** instance to stream *os*.

B.2.26. sodl::StartSimulation

A ***sodl::StartSimulation*** message is sent to all processes to begin the simulation run. Its time stamp is set to ***Clock::getStartTime***(), which defaults to -1.0. This value allows a significant amount of simulation initialization prior to time 0.

Parent Class: **public** ***sodl::SystemMessage***

Derived Classes: None

Public Constructors:

sodl::StartSimulation::StartSimulation*(void)** – This constructor calls the parent constructor by ***SystemMessage::SystenMessage(0,0,***sodl::Defs::SMT_StartSimulation***). It also sets its time stamp value to ***Clock::getStartTime***().

Public Methods:

static void ***sodl::StartSimulation::typeInit***(***sodl::*mtype** *t*) – Called to initialize portions of the ***std::vector*<bool>** ***sodl::Defs::msgTypes***.

B.2.27. sodl::SystemMessage

There are a number of system-defined messages that manage various aspects of the SODL run-time system. These system-defined messages are all derived from the ***sodl::SystemMesage*** class.

Parent Class: **public** ***sodl::Message***

Derived Classes: ***sodl::AntiMessage***, ***sodl::EndSimulation***, ***sodl::StartSimulation***, ***sodl::UpdateGVT***

Protected Constructors:

sodl::SystemMessage::SystemMessage**(ulong** *n*, **ulong** *i*, ***sodl*****::mtype** *t***)** – This constructor calls the parent class constructor ***sodl::Message*****(***n*, *i*, *t***)**.

sodl::SystemMessage::SystemMessage**(const process&** *p*, **message** *h*, **mtype** *ty*, **double** *t***)** – This constructor calls the parent class constructor ***sodl::Message*****(***p*, *h*, *ty*, *t***)**.

Public Methods:

static void ***sodl::SystemMessage::typeInit*****(*****sodl*****::mtype** *t***)** – Called to initialize portions of the ***std::vector*****<bool>** ***sodl::Defs::msgTypes***.

bool ***sodl::SystemMessage::getTX*****(void)** – Overloaded to always return **true**.

B.2.28. sodl::TextViewManager

This is the default view manager for a simulation engine. It is a bare bones manager providing no support beyond handling idle events.

Parent Classes: **public** ***sodl::ViewManager***

Derived Classes: None

Public Constructors:

sodl*::*TextViewManager*::*TextViewManager*(*sodl*::*IdleListener*&** *l*, **int *argc*, **char*[]** *argv*) – This constructor will invoke the parent class constructor by calling ***sodl*::*ViewManager*(*l*, *argc*, *argv*)**. In this case, both *argc* and *argv* are ignored. They are retained primarily for the benefit of other classes derived from ***sodl*::*ViewManager***.

Public Methods:

virtual void *sodl*::*TextViewManager*::*start*(void) – Calls ***idleListener.start*()** followed by repeatedly calling ***idleListener.idle*()** until it returns **false**.

B.2.29. sodl::TimeStamp

This ***sodl*::*TimeStamp*** class provides a mechanism for time stamping items within the simulation system. These items requiring time stamps are messages, processes, and some other internal classes. Each time stamp value is with reference to a specific ***sodl*::*Engine*** instance. For a given process, this engine is the one that controls the process. For messages, it is the engine where the message will eventually be delivered. Messages also have a generation time that is relative to the engine where the message originated.

Parent Classes: None

Derived Classes: ***sodl*::*Clock***, ***sodl*::*Message***, ***sodl*::*Process***, ***sodl*::*ProcessMode***

Private Data Members:

double *sodl*::*TimeStamp*::*time* – Actual time value of the time stamp.

ulong *sodl*::*TimeStamp*::*node* – Index of the engine to which the time is relative

Public Constructors:

sodl*::*TimeStamp*::*TimeStamp*(double** *t*, **ulong** *n*) – Sets ***time to *t* and ***node*** to *n*.

Public Methods:

virtual double ***sodl::TimeStamp::getTime*(void) const** – Returns to the calling routine ***sodl::TimeStamp::time***.

virtual ***sodl::Engine& sodl::TimeStamp::getEngine*(void)** – Returns to the calling routine ***sodl::EngineStand::stand*[*node*]**.

virtual ulong ***sodl::TimeStamp::getNode*(void) const** – Returns to the calling routine ***node***.

virtual void ***sodl::TimeStamp::setTime*(double** *t***)** – Sets ***time*** to *t*.

virtual void ***sodl::TimeStamp::setEngine*(const** ***sodl::Engine&*** *e***)** – Sets ***node*** to ***e.getNode*()**.

virtual void ***sodl::TimeStamp::setEngine*(ulong** *n***)** – Sets ***node*** to *n*.

B.2.30. sodl::Trace

The ***sodl::Trace*** class is used to perform procedure call tracing. It requires a bit of effort to set up, but the entire SODL run-time system has instrumentation to trace program execution and log the execution to a file if desired. This is done through a stack mechanism. Upon entering a procedure, the programmer calls the ***enter*(...)** method to log the fact that the entry occurred. Immediately prior to leaving the routine, the programmer makes a call to ***leave***(...) with the same parameters as the matching ***leave***(...).Trace will note any discrepancies. Programmers can also turn on and off tracing in specific routines without regard to any other part of the program.

We should note that the ***sodl::Trace*** class member data and methods are only defined when the macro ***_TRACE*** is defined. Programmers wishing to make use of the routines here will need to delimit their calls with "**#ifdef *_TRACE***" ... "**#endif**".

Parent Class: None

Derived Classes: ***sodl::Defs***, ***gvm::Object***, ***gvm::View***.

Private Data Members:

static ***std*::*ostream** *sodl*::*Trace*::*active*** – Pointer to the currently active output stream.

static ***std*::*stack*<*std*::*pair*<bool, *std*::*string*> >* *sodl*::*Trace*::*calls*** - Stack containing the call trace of the program. The ***calls.top().first*** controls where the output is directed, */dev/null* for **false**, and *trace.log* for **true**.

static ulong ***sodl*::*Trace*::*indentCount*** – The count of the number of pairs in the ***calls*** stack that have **true** for the ***first*** component of the stack element. This is used for computing how far to indent each line of output the ***sodl*::*Trace*** class methods produce.

static ***std*::*ofstream** *sodl*::*Trace*::*null*** – A pointer to an output file stream which dumps the output to */dev/null.*

static ***std*::*ofstream** *sodl*::*Trace*::*trace*** – A pointer to an output file stream directing output to the local file *trace.log*.

Private Methods:

static void ***sodl*::*Trace*::*staticInit*(void)** – Allocates the various data structures described above, and sets ***active*** to direct output to *trace.log* as the default. It also initializes ***indentCount*** to 0.

Public Methods:

static void ***sodl*::*Trace*::*enter*(bool** ***d*, *std*::*string*** *s*) – This method should be called upon entry into a block of code, normally a function or class method. The input parameters *d* and *s* are paired and pushed onto the top of ***calls***. If the input parameter *d* is **true**, ***active*** is set to direct output to *trace.log* and it increments by 2 ***indentCount***; if d is false, ***active*** directs output to */dev/null.* The string *s* is then formatted and sent to the ***active*** output stream

static ***std*::*string*** ***sodl*::*Trace*::*indent*(void)** – This routine returns a string of spaces ***indentCount*** characters long.

static void ***sodl*::*Trace*::*leave*(bool** *d*, ***std*::*string*** *s*) – This method should be called immediately prior to leaving a block of code, typically a function or class method that had a corresponding call to ***enter*(...)** earlier in the program's execution. The input parameters, when paired together, should match the top element of the call stack, ***calls***. This is checked to ensure that ***enter*()** and ***leave*(...)** statements are properly paired. The input parameter *s* is appended to an indented line (as generated in ***indent*()**) and sent to the currently active output stream. If *d* is **true**, then ***indentCount*** is decremented by 2. The top element of the calls stack is removed and the new top element is evaluated to determine where the active stream should now point… *trace.log* if ***calls.top().first*** is **true**, */dev/null* otherwise.

static ***std*::*ostream*& *sodl*::*Trace*::*line*(void)** – This produces a line in the currently active log file which is indented and prefaced with "—". The return stream is the active stream.

static ***std*::*ostream*& *sodl*::*Trace*::*tlog*(void)** – This simply returns *** *active*** so that the calling routine may provide non-indented text to the currently active stream.

static void ***sodl*::*Trace*::*stackTrace*(*std*::*ostream*&** *os*) – This routine makes a copy of the current ***calls*** stack and dumps that copy to the stream *os*.

B.2.31. sodl::UpdateGVT

The ***sodl::UpdateGVT*** message class is intended to act as a catalyst for computation of the Global Virtual Time (GVT). It notifies the various ***sodl*::*EngineStand*** instances in a distributed simulation to begin computing the GVT.

AS OF THIS WRITING, THIS CLASS IS NOT USED.

Parent Classes: **public** ***SystemMessage***

Derived Classes: None

B.2.32. sodl::ViewManager

The view manager is an abstract class declaration, the subclasses of which are intended to manage certain aspects of the IO operations between the user and simulation engine. It requires a ***sodl::IdleListener*** instance, which is normally a ***sodl::EngineStand*** instance. When a simulation system is produced using the SODL system, exactly one ***sodl::ViewManager*** instance is created on each node in the distributed simulation.

Parent Classes: **public *sodl::Defs***

Derived Classes: ***sodl::GLUTViewManager***, ***sodl::TextViewManager***.

Protected Data Members:

sodl::IdleListener& sodl::ViewManager::idleListener – During idle times the method ***idleListener.idle***()is called, allowing the simulation to process some of the pending messages.

Public Constructors:

sodl::ViewManager::ViewManager(***sodl::IdleListener&*** *l*, **int*** *argc*, **char**** *argv*)– Sets ***idleListener*** to *l*. The remaining arguments are for sub classes to perform initialization from command line parameters.

Public Methods:

virtual void *sodl::ViewManager::start*(void)=0 – This function is intended to be overloaded by a subclass. Its functionality should include at a minimum performing last minute initialization and call the ***idleListener.start***() method. It also starts some mechanism whereby ***idleListener.idle***() is repeatedly called to allow the simulation to proceed.

B.2.33. SODL run-time system items not associated with a specific class

Parent Classes: N/A

Derived Classes: N/A

Typedefs:

typedef unsigned char ::byte – Shorthand for unsigned char.

typedef unsigned int ::uint – Shorthand for unsigned int.

typedef unsigned long ::ulong – Shorthand for unsigned long.

typedef *sodl*::*MessageHandle* *sodl*::message – Shorthand for ***sodl*::*MessageHandle*** class instance as they apply to ***sodl*::*Message*** instances.

typedef *sodl*::*ProcessHandle* *sodl*::process – Shorthand for ***sodl*::*ProcessHandle*** class instance as they apply to ***sodl*::*Process*** instances.

typedef *sodl*::*Defs*::*MessageType* *sodl*::mtype – This typedef is for specifying a shorthand notation for the ***sodl*::*MiniProcess*::*MessageType*** enumerator.

typedef *sodl*::*ProfileTools* *sodl*::profile – Shorthand for a ***sodl*::*ProfileTools*** class instance.

typedef *sodl*::*Defs*::*ProcessType* *sodl*::ptype – This typedef is for specifying a shorthand notation for the ***sodl*::*MiniProcess*::*ProcessType*** enumerator.

typedef *sodl*::*Random* *sodl*::rand – Shorthand for a ***sodl*::*Random*** class instance.

typedef *std*::*priority_queue*<*sodl*::*ScheduleItem*, *std*::*vector*<*sodl*::*ScheduleItem*>, *std*::*greater*<*sodl*::*ScheduleItem*> > *sodl*::schedule – This provides a convenience declaration for a schedule of fossil collection events.

Functions:

std*::*string **::*alpha*(const bool** *v*) – This returns the string "true" when *v* is **true** and "false" when *v* is **false**.

template<class T> T *dot*(***std*::*valarray*<T>** *x*, ***std*::*valarray*<T>** *y*) – Returns the dot product of *x* and *y*.

int ::*main*(int *argc*, **char** **argv*[]) – Start point for the simulation program execution.

template<class T> ***std*****::*****vector*****<T> ::*****make_vector*****(ulong** *s*, **T** *v*, **...)** – This routine will create new vector of size *s* with the parameters starting with *v*.

template <class T> T ::***max*****(T** *x*, **T** *y*) – Returns the maximum value of *x* and *y*. Type T must have defined the operator '<'.

template <class T> T ::***min*****(T x, T y)** – Returns the minimum value of *x* and *y*. Type T must have defined the operator '<'.

template<class T> T ***norm*****(*****std*****::*****valarray*****<T>** *x*) – Returns the 2-norm of *x* (i.e. ***sqrt*****(*****dot*****(***x*, *x*)).

std**::*****ostream*****& ::*****operator<<*****(*****std*****::*****ostream*****&** *os*, **const** ***sodl*****::*****Defs*****&** *v*) – This routine calls v.***serialize***(*os*).

std**::*****ostream*****& ::*****operator<<*****(*****std*****::*****ostream*****&** *os*, **const** ***sodl*****::*****Defs******** *v*) – This routine calls (*v).***serialize***(*os*).

std**::*****ostream*****& ::*****operator<<*****(*****std*****::*****ostream*****&** *os*, **const** ***sodl*****:: mtype** *t*) – This routine sends to the output stream *os* a string representation of *t* given by the value ***sodl*****::*****Defs*****::*****msgNames***[*t*].

std**::*****ostream*****& ::*****operator<<*****(*****std*****::*****ostream*****&** *os*, **const** ***sodl*****::ptype** *t*) – This routine sends to the output stream *os* a string representation of *t* given by the value ***sodl*****::*****Defs*****::*****procNames***[*t*].

std**::*****ostream*****& ::*****operator<<*****(*****std*****::*****ostream*****&** *os*, **const** ***sodl*****::*****Exception*****::*****Nonspecific*****&** *e*) – This routine calls ***e.serialize***(*os*) to produce output related to an exception.

template <class T, class A> ***std*****::*****ostream*****& ::*****operator<<*****(*****std*****::*****ostream*****& os, const** ***std*****::*****deque*****<T,A>&** *v*) – Produces formatted textual output of the components of a ***std::deque*** class instance. There must be a **::*****operator<<*****(*****std*****::*****ostream*****&, const T&)** declared somewhere allowing each component in the container to be displayed.

template <class T, class A> ***std*****::*****ostream*****& ::*****operator<<*****(*****std*****::*****ostream*****&** *os*, **const** ***std*****::*****list*****<T,A>&** *v*) – Produces formatted textual output of the components of a ***std::list*** class instance. The operator **::*****operator<<*****(*****std*****::*****ostream*****&, const T&)** must be declared somewhere allowing each component in the container to be displayed.

template <class K, class T, class C, class A> ***std*::*ostream*&** ***::operator<<*(*std*::*ostream*&** *os*, **const** ***std*::*map*<K,T,C,A>&** *v*) – Produces formatted textual output of the components of a ***std*::*map*** class instance. The operators ***::operator<<*(*std*::*ostream*&, const K&)** and ***::operator<<*(*std*::*ostream*&, const T&)** must both be declared somewhere allowing each component in the container to be properly displayed. The output is sorted according to the sort order specified by the declaration of *v*.

template <class K, class T, class C, class A> ***std*::*ostream*&** ***::operator<<*(*std*::*ostream*&** *os*, **const** ***std*::*multimap*<K,T,C,A>&** *v*) – Produces formatted textual output of the components of a ***std*::*multimap*** class instance. The operators ***::operator<<*(*std*::*ostream*&, const K&)** and ***::operator<<*(*std*::*ostream*&, const T&)** must both be declared somewhere allowing each component in the container to be properly displayed. The output is sorted according to the sort order specified by the declaration of *v*.

template <class T, class C, class A> ***std*::*ostream*&** ***::operator<<*(*std*::*ostream*&** **os, const** ***std*::*multiset*<T,C,A>&** *v*) – Produces formatted textual output of the components of a ***std*::*multiset*** class instance. The operator ***::operator<<*(*std*::*ostream*&, const T&)** must be declared somewhere allowing each component in the container to be displayed. The output is sorted according to the sort order specified by the declaration of *v*.

template <class T1, class T2> ***std*::*ostream*&** ***::operator<<*(*std*::*ostream*&** *os*, **const** ***std*::*pair*<T1, T2>&** *v*) – Produces formatted textual output of a ***std*::*pair*** class instance. ***::operator<<*(*std*::*ostream*&, const T1&)** and ***::operator<<*(*std*::*ostream*&, const T2&)** must both be declared allowing each component to be properly displayed.

template <class T, class C, class L> ***std*::*ostream*&** ***::operator<<*(*std*::*ostream*&** *os*, **const** ***std*::*priority_queue*<T,C,L>&** *v*) – Produces formatted textual output of a ***std*::*priority_queue*** class instance. The operator ***::operator<<*(*std*::*ostream*&, const T&)** must be declared somewhere allowing each component in the container to be properly displayed. Since there is no iterator for ***std*::*priority_queue*** instances, *v* is copied and items are printed from the top of this copied priority queue prior to their removal.

template <class T, class C> ***std*::*ostream*&** ***::operator<<*(*std*::*ostream*&** *os*, **const** ***std*::*queue*<T,C>&** *v*) – Produces formatted textual output of the components of a ***std*::*stack*** class instance. The operator

::***operator***<<(***std***::***ostream***&, **const T&)** must be declared somewhere allowing each component in the container to be properly displayed. Since there is no iterator for ***std***::***queue*** instances, *v* is copied and items are printed from the top of this copied queue prior to their removal.

template <class T, class C, class A> ***std***::***ostream***& ::***operator***<<(***std***::***ostream***& *os*, **const** ***std***::***set***<T,C,A>& v) – Produces formatted textual output of the components of a ***std***::***set*** class instance. The operator ::***operator***<<(***std***::***ostream***&, **const T&)** must be declared somewhere allowing each component in the container to be displayed. The output is sorted according to the sort order specified by the declaration of *v*.

template <class T, class C> ***std***::***ostream***& ::***operator***<<(***std***::***ostream***& *os*, **const** ***std***::***stack***<T,C>& *v*) - T) – Produces formatted textual output of the components of a ***std***::***stack*** class instance. The operator ::***operator***<<(***std***::***ostream***&, **const T&)** must be declared somewhere allowing each component in the container to be properly displayed. Since there is no iterator for ***std***::***stack*** instances, *v* is copied and items are printed from the top of this copied stack prior to their removal.

template <class T> ***std***::***ostream***& ::***operator***<<(***std***::***ostream***& *os*, **const** ***std***::***valarray***<T >& *v*) – Produces formatted textual output of the components of a ***std***::***valarray*** class instance. The operator ::***operator***<<(***std***::***ostream***&, **const T&)** must be declared somewhere allowing each component in the container to be displayed.

template <class T, class A> ***std***::***ostream***& ::***operator***<<(***std***::***ostream***& *os*, **const** ***std***::***vector***<T,A>& *v*) – Produces formatted textual output of the components of a ***std***::***vector*** class instance. The operator ::***operator***<<(***std***::***ostream***&, **const T&)** must be declared somewhere allowing each component in the container to be displayed.

bool ::***operator***>(**const** ***sodl***::***Handle***& *h1*, **const** ***sodl***::***Handle***& *h2*) – This routine compares two ***sodl***::***Handle*** instances and returns **true** exactly when *h1*.***getNode***() > *h2*.***getNode***(), or when *h1*.***getNode***() = *h2*.***getNode***() and *h1*.***getIndex***() > *h2*.***getIndex***().

bool ::*operator*>(const *sodl*::*Message*& *m1*, const *sodl*::*Message*& *m2*) – This routine compares two ***sodl*::*Message*** instances and returns **true** exactly when *m1*.***getTime***() > *m2*.***getTime***(), or when *m1*.***getTime***() = *m2*.***getTime***() and *m1*.***getID***() > *m2*.***getID***().

bool ::*operator*<(const *sodl*::*Handle*& *h1*, const *sodl*::*Handle*& *h2*) – This routine compares two ***sodl*::*Handle*** instances and returns **true** exactly when *h1*.***getNode***() < *h2*.***getNode***(), or when *h1*.***getNode***() = *h2*.***getNode***() and *h1*.***getIndex***() < *h2*.***getIndex***().

bool ::*operator*<(const *sodl*::*Message*& *m1*, const *sodl*::*Message*& *m2*) – This routine compares two ***sodl*::*Message*** instances and returns **true** exactly when *m1*.***getTime***() < *m2*.***getTime***(), or when *m1*.***getTime***() = *m2*.***getTime***() and *m1*.***getID***() < *m2*.***getID***().

bool ::*operator*>=(const *sodl*::*Handle*& *h1*, const *sodl*::*Handle*& *h2*) – This routine compares two ***sodl*::*Handle*** instances and returns **true** exactly when *h1*.***getNode***() > *h2*.***getNode***(), or when *h1*.***getNode***() = *h2*.***getNode***() and *h1*.***getIndex***() ≥ *h2*.***getIndex***().

bool ::*operator*>=(const *sodl*::*Message*& *m1*, const *sodl*::*Message*& *m2*) – This routine compares two ***sodl*::*Message*** instances and returns **true** exactly when *m1*.***getTime***() > *m2*.***getTime***(), or when *m1*.***getTime***() = *m2*.***getTime***() and *m1*.***getID***() ≥ *m2*.***getID***().

bool ::*operator*<=(const *sodl*::*Handle*& *h1*, const *sodl*::*Handle*& *h2*) – This routine compares two ***sodl*::*Handle*** instances and returns **true** exactly when *h1*.***getNode***() < *h2*.***getNode***(), or when *h1*.***getNode***() = *h2*.***getNode***() and *h1*.***getIndex***() ≤ *h2*.***getIndex***().

bool ::*operator*<=(const *sodl*::*Message*& *m1*, const *sodl*::*Message*& *m2*) – This routine compares two ***sodl*::*Message*** instances and returns **true** exactly when *m1*.***getTime***() < *m2*.***getTime***(), or when *m1*.***getTime***() = *m2*.***getTime***() and *m1*.***getID***() ≤ *m2*.***getID***().

bool ::*operator*==(const *sodl*::*Handle*& *h1*, const *sodl*::*Handle*& *h2*) – This routine compares two ***sodl*::*Handle*** instances and returns **true** exactly when *h1*.***getNode***() = *h2*.***getNode***() and *h1*.***getIndex***() = *h2*.***getIndex***().

bool ::*operator*==(**const** ***sodl*::*Message*&** *m1*, **const** ***sodl*::*Message*&** *m2*) – This routine compares two ***sodl*::*Message*** instances and returns **true** exactly when *m1*.***getTime***() = *m2*.***getTime***() and *m1*.***getID***() = *m2*.***getID***().

bool ::*operator!*=(**const** ***sodl*::*Handle*&** *h1*, **const** ***sodl*::*Handle*&** *h2*) – This routine compares two ***sodl*::*Handle*** instances and returns **true** exactly when *h1*.***getNode***() ≠ *h2*.***getNode***() or *h1*.***getIndex***() ≠ *h2*.***getIndex***().

bool ::*operator!*=(**const** ***sodl*::*Message*&** *m1*, **const** ***sodl*::*Message*&** *m2*) – This routine compares two ***sodl*::*Message*** instances and returns **true** exactly when *m1*.***getTime***() ≠ *m2*.***getTime***() and *m1*.***getID***() ≠ *m2*.***getID***().

template<class T> ***std*::*vector*<T>** ::*resize_vector*(**int** *s*, **T** *d*, ***std*::*vector*<T>** *v*) – This routine will create a copy of the ***std*::*vector*** *v*, resized to size *s*, and padded with the value *d* in the case that *v* needs to grow.

void ::*staticInit*(**int*** *argc*, **char*** *argv*[]) – Performs static variable initialization particularly associated with type information in ***sodl*::*Process*** and ***sodl*::*Message*** class definitions. It also initializes the view manager instance.

B.3. SODL - GLUT interface

There is a collection of SODL constructs used with the ***sodl*::*GLUTViewManager*** allowing SODL programs to display information to a window that is not located on the local host.

This interface is implemented as a scene graph. For more detailed information on the notions of a scene graph, refer to (Foley 1996). Simulation system developers can have multiple views, each of which displays things completely independently. Each view owns a collection of graphics nodes each of which having an associated affine transformation. Each of these graphics nodes can themselves have a collection of child graphics nodes (again with their associated affine transformation) and so on. Child nodes inherit the affine transformation of their parent by which they multiply their own so that all of its child nodes inherit the new combined transformation. Additionally, nodes can have a collection of subordinate shapes with different properties (color, rendering mode, etc). The aggregate affine transformation associated with the owner node is applied to the shapes in order to display them in the proper orientation, scale and location within the rendered scene.

B.3.1. message:AddNode

This serves as parent class to the two **message:*AddNode*** derivatives below. It serves little more purpose than to act as a type placeholder.

Parent Message Construct: **message:*AddSubordinate***

Derived Message Constructs: **message:*AddNode2D***, **message:*AddNode3D***

Receiving Processes: None

Sending Processes: None

B.3.2. message:AddNode2D

This message is used to add multiple **process:*Node2D*** instances to a parent **process:*Node2D*** or **process:*View2D***. Those receiving processes add as subordinate nodes those listed in ***AddSubordinate::subrdinates***.

Parent Message Construct: **message:*AddNode***

Derived Message Constructs: None

Receiving Processes: **process:*Node2D***, **process:*View2D***

Sending Processes: None

B.3.3. message:AddNode3D

This message is used to add multiple **process:*Node3D*** instances to a parent **process:*Node3D*** or **process:*View3D***. Those receiving processes add as subordinate nodes those listed in ***AddSubordinate::subrdinates***.

Parent Message Construct: **message:*AddNode***

Derived Message Constructs: None

Receiving Processes: **process:*Node3D***, **process:*View3D***

Sending Processes: None

B.3.4. message:AddShape

The **message:*AddShape*** construct serves as parent to **message:*AddShape2D*** and **message:*AddShape3D***. It serves little more purpose than to act as a type placeholder.

Parent Message Construct: **message:*AddSubordinate***

Derived Message Constructs: **message:*AddShape2D***, **message:*AddShape3D***

Receiving Processes: None

Sending Processes: None

B.3.5. message:AddShape2D

This message is used to add multiple **process:*Shape2D*** instances to a parent **process:*Node2D***. Those receiving processes add as subordinate shapes those listed in ***AddSubordinate::subrdinates***.

Parent Message Construct: **message:*AddShape***

Derived Message Constructs: None

Receiving Processes: **process:*Node2D***

Sending Processes: None

B.3.6. message:AddShape3D

This message is used to add multiple **process:*Shape3D*** instances to a parent **process:*Node3D*.** Those receiving processes add as subordinate shapes those listed in ***AddSubordinate*::*subrdinates***.

Parent Message Construct: **message:*AddShape***

Derived Message Constructs: None

Receiving Processes: **process:*Node3D***

Sending Processes: None

B.3.7. message:AddSubordinate

The **message:*AddSubordinate*** construct provides a common mechanism for adding subordinate process references to a parent process. It contains an array of **process** instances that identify these subordinate processes. These messages are sent to processes to have the members listed in ***subordinates***.

Parent Message Construct: None

Derived Message Constructs: **message:*AddNode***, **message:*AddShape***, **message:*AddVertex***, and **message:*Register***.

Receiving Processes: None

Sending Processes: None

Data Members:

process:*subordinates*[] – An array of process handles that are intended to be added to a subordinate list managed by the recipient.

Methods:

method:*add*(public; void; process:*n*;) – Adds the process *n* to back of the ***subordinates*** vector.

method:*getTX*(public; bool;) – An overload of the ***sodl*::*Message*::*getTX*()**. It will return **true** exactly when subordinates is not empty and ***sodl*::*Message*::*getTX*()** is **true**.

method:*size*(public; ulong;) – Returns ***subordinates.size*()** to the calling routine.

B.3.8. message:AddVertex

The **message:*AddVertex*** construct serves as parent to **message:*AddVertex2D*** and **message:*AddVertex3D***. It serves little more purpose than to act as a type placeholder.

Parent Message Construct: **message:*AddSubordinate***

Derived Message Constructs: **message:*AddVertex2D***, **message:*AddVertex3D***

Receiving Processes: None

Sending Processes: None

B.3.9. message:AddVertex2D

This message is used to add multiple **process:*Vertex2D*** instances to a parent **process:*Shape2D***. Those receiving processes add as subordinate shapes those listed in ***AddSubordinate*::*subrdinates***.

Parent Message Construct: **message:*AddShape***

Derived Message Constructs: None

Receiving Processes: **process:*Shape2D***

Sending Processes: None

B.3.10. message:AddVertex3D

This message is used to add multiple **process:*Vertex3D*** instances to a parent **process:*Shape3D***. Those receiving processes add as subordinate shapes those listed in ***AddSubordinate*::*subrdinates***.

Parent Message Construct: **message:*AddShape***

Derived Message Constructs: None

Receiving Processes: **process:*Shape3D***

Sending Processes: None

B.3.11. message:AddView

This message is used to inform GUI objects that they have just been added to a view. They normally are generated by the view after the GUI object's parent registered it with the view.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*Cone***, **process:*Cube***, **process:*Node***, **process:*Node2D***, **process:*Node3D***, **process:*Object***, **process:*Polygon2D***, **process:*Polygon3D***, **process:*Shape***, **process:*Sphere***, **process:*Torus***, **process:*Vertex2D***, **process:*Vertex3D***

Sending Processes: **process:*View2D***, **process:*View3D***

B.3.12. message:RefreshDisplay

This message causes a recipient **process:*View*** instance to refresh its display.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*View***

Sending Processes: **process:*View***

B.3.13. message:Register

Register messages register graphics components with the view that is their parent. Normally a node will register its child nodes and shapes. Polygons register their vertices. The objects being registered are stored in the ***AddSubordinate::subrdinates***.

Parent Message Construct: **message:*AddSubordinate***

Derived Message Constructs: **message:*RegisterNode***, **message:*RegisterShape***, **message:*RegisterVertex***

Receiving Processes: None

Sending Processes: None

Data Members:

gvm::object_index:index*((ulong) –1)** – Stores object index of the sender with respect to the ***gvm::View instance that will eventually get the message derived from this construct.

B.3.14. message:RegisterNode

This construct acts mainly as the type placeholder and parent construct for the two derived messages.

Parent Message Construct: **message:*Register***

Derived Message Constructs: **message:*RegisterNode2D***, **message:*RegisterNode3D***

Receiving Processes: None

Sending Processes: None

B.3.15. message:RegisterNode2D

This message registers **process:*Node2D*** instances with a **process:*View2D***. The process handles are listed in ***AddSubordinate::subrdinates***.

Parent Message Construct: **message:*RegisterNode***

Derived Message Constructs: None

Receiving Processes: **process:*View2D***

Sending Processes: **process:*Node2D***

B.3.16. message:RegisterNode3D

This message registers **process:*Node3D*** instances with a **process:*View3D***. The process handles are listed in ***AddSubordinate*::*subrdinates***.

Parent Message Construct: **message:*RegisterNode***

Derived Message Constructs: None

Receiving Processes: **process:*View3D***

Sending Processes: **process:*Node3D***

B.3.17. message:RegisterShape

This construct acts mainly as the type placeholder and parent construct for the two derived messages.

Parent Message Construct: **message:*Register***

Derived Message Constructs: **message:*RegisterShape2D***, **message:*RegisterShape3D***

Receiving Processes: None

Sending Processes: None

B.3.18. message:RegisterShape2D

This message registers **process:*Shape2D*** instances with a **process:*View2D***. The process handles are listed in ***AddSubordinate*::*subrdinates***.

Parent Message Construct: **message:*RegisterShape***

Derived Message Constructs: None

Receiving Processes: **process:*View2D***

Sending Processes: **process:*Node2D***

B.3.19. message:RegisterShape3D

This message registers **process:*Shape3D*** instances with a **process:*View3D***. The process handles are listed in ***AddSubordinate*::*subrdinates***.

Parent Message Construct: **message:*RegisterShape***

Derived Message Constructs: None

Receiving Processes: **process:*View3D***

Sending Processes: **process:*Node3D***

B.3.20. message:RegisterVertex

This construct acts mainly as the type placeholder and parent construct for the two derived messages.

Parent Message Construct: **message:*Register***

Derived Message Constructs: **message:*RegisterVertex2D***, **message:*RegisterVertex3D***

Receiving Processes: None

Sending Processes: None

B.3.21. message:RegisterVertex2D

This message registers **process:*Vertex2D*** instances with a **process:*View2D***. The process handles are listed in ***AddSubordinate*::*subrdinates***.

Parent Message Construct: **message:*RegisterShape***

Derived Message Constructs: None

Receiving Processes: **process:*View2D***

Sending Processes: **process:*Shape2D***

B.3.22. message:RegisterVertex3D

This message registers **process:*Vertex3D*** instances with a **process:*View3D***. The process handles are listed in ***AddSubordinate*::*subrdinates***.

Parent Message Construct: **message:*RegisterShape***

Derived Message Constructs: None

Receiving Processes: **process:*View3D***

Sending Processes: **process:*Shape3D***

B.3.23. message:SelectiveActivate

This message is delivered to a process and allows the respective ***gvm*::*Object*** instances controlled by different ***gvm*::*View*** instances to be selectively activated or deactivated. A **process:*Object*** instance, upon receiving a **message:*SelectiveActivate*** will send a **message:*SetActive*** to each ***views***[*i*] with the active flag set to ***active***[*i*].

Parent Message Construct: None

Derived Message Constructs: None

Receiving Processes: **process:*Object***

Sending Processes: None

Data Members:

process:*views*[] – This is the list of **process:*View*** instances affected by the change in the active flag.

bool:*active*[] – Each element in this array sets the active flag in the associated **process:*View*** for the ***gvm*::*Object*** corresponding to this **process:*Object***.

Methods:

method:*add*(public; void; process:*v*; bool:*a*;) – This calls methods ***views.push_back*(*v*)** and ***active.push_back*(*a*)** to insert the view and active flag for that view into their respective vectors.

method:*getView*(public; process; ulong:*i*;) – This will return the *i*th process instance in the view vector (i.e. ***view***[*i*]).

method:*getActive*(public; bool; ulong:*i*;) – This will return the *i*th process instance in the active vector (i.e. ***active***[*i*]).

method:*size*(public; ulong;) – This method returns the number of elements in the ***view*** array (i.e. ***view.size***()). This should also be the number of elements in the ***active*** array, and will be if **method:*add*** is used instead of manually adding elements to the arrays.

B.3.24. message:SetActive

Upon receipt of this message, a **process:*Object*** instance will set its active flag to that of the active variable in the message payload. It will then broadcast additional **message:*SetActive*** instances to the views with which the **process:*Object*** instance has been registered.

Parent Message Construct: **message:*SetDefaultActive***

Derived Message Constructs: None

Receiving Processes: **process:*Object***, **process:*View***

Sending Processes: **process:*Object***

B.3.25. message:SetAffine

This sets various affine transform values. These allow graphical objects to be moved within a scene.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: **message:*SetAffine2D***, **message:*SetAffine3D***

Receiving Processes: None

Sending Processes: None

Data Members:

GLdouble:*ctrRot*[] – Specifies a center of rotation.

GLdouble:*ctrScale*[] – Specifies a scaling center.

GLdouble:*rot*[] – Specifies a rotation angle about each axis.

GLdouble:*scale*[] – Specifies a scaling factor along each axes.

GLdouble:*trans*[] – Specifies a translation factor along each axes.

Methods:

method:*set*(public; void; *std*::*vector*<GLdouble>:*cr*; *std*::*vector*<GLdouble>:*cs*; *std*::*vector*<GLdouble>:*r*; *std*::*vector*<GLdouble>:*s*; *std*::*vector*<GLdouble>:*t*;) – This sets ***ctrRot***, ***ctrScale***, ***rot***, ***scale***, and ***trans*** to *cr*, *cs*, *r*, *s*, and *t* respectively, using **::*resize_vector*(…)** to copy the values and ensure that the size of the vectors remains unchanged. That is, if this message is implemented as a **message:*SetAffine2D*** instance, each of the arrays has size 2. The **::*resize_vector*(…)** function is used to ensure that this size remains fixed after the assignment.

B.3.26. message:SetAffine2D

Specializes **message:*SetAffine*** for 2D affine transformations.

Parent Message Construct: **message:*SetAffine***

Derived Message Constructs: None

Receiving Processes: **process:*Node2D***, **process:*View2D***

Sending Processes: **process:*Node2D***

Methods:

method:*init*(public; void;) – Initializes the data members so that the aggregate affine transform is the identity. This involves resizing all of the data members to 2 elements with values 0.0 (except ***SetAffine*::*scale***, which is initialized to <1.0, 1.0>).

method:*setCtrRotation*(public; void; GLdouble:*x*; GLdouble:*y*;) – This method sets the value of ***SetAffine*::*ctrRot*** to **::*make_vector***(2, *x*, *y*).

method:*setCtrRotation*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*ctrRot*** to **::*resize_vector***(2, 0.0, *v*).

method:*setCtrScale*(public; void; GLdouble:*x*; GLdouble:*y*;) – This method sets the value of ***SetAffine*::*ctrScale*** to **::*make_vector***(2, *x*, *y*).

method:*setCtrScale*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*ctrScale*** to **::*resize_vector***(2, 0.0, *v*).

method:*setRotation*(public; void; GLdouble:*x*; GLdouble:*y*;) – This method sets the value of ***SetAffine*::*rot*** to **::*make_vector***(2, x, y).

method:*setRotation*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*rot*** to **::*resize_vector***(2, 0.0, *v*).

method:*setScale*(public; void; GLdouble:*x*; GLdouble:*y*;) – This method sets the value of ***SetAffine*::*scale*** to **::*make_vector***(2, x, y).

method:*setScale*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*scale*** to **::*resize_vector***(2, 1.0, *v*).

method:*setTranslation*(public; void; GLdouble:*x*; GLdouble:*y*;) – This method sets the value of ***SetAffine*::*trans*** to **::*make_vector***(2, x, y).

method:*setTranslation*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*trans*** to **::*resize_vector***(2, 0.0, *v*).

B.3.27. message:SetAffine3D

Specializes **message:*SetAffine*** for 3D affine transformations.

Parent Message Construct: **message:*SetAffine***

Derived Message Constructs: None

Receiving Processes: **process:*Node3D***, **process:*View3D***

Sending Processes: **process:*Node3D***

Methods:

method:*init*(public; void;) – Initializes the data members so that the aggregate affine transform is the identity. This involves resizing all of the data members to 3 elements with values 0.0 (except ***SetAffine*::*scale***, which is initialized to <1.0, 1.0, 1.0>).

method:*setCtrRotation*(public; void; GLdouble:*x*; GLdouble:*y*; GLdouble:*z*;) – This method sets the value of ***SetAffine*::*ctrRot*** to **::*make_vector***(3, *x*, *y*, *z*).

method:*setCtrRotation*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*ctrRot*** to **::*resize_vector***(3, 0.0, *v*).

method:*setCtrScale*(public; void; GLdouble:*x*; GLdouble:*y*; GLdouble:*z*;) – This method sets the value of ***SetAffine*::*ctrScale*** to **::*make_vector***(3, *x*, *y*, *z*).

method:*setCtrScale*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*ctrScale*** to **::*resize_vector***(3, 0.0, *v*).

method:*setRotation*(public; void; GLdouble:*x*; GLdouble:*y*; GLdouble:*z*;) – This method sets the value of ***SetAffine*::*rot*** to **::*make_vector***(3, *x*, *y*, *z*).

method:*setRotation*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*rot*** to **::*resize_vector***(3, 0.0, *v*).

method:*setScale*(public; void; GLdouble:*x*; GLdouble:*y*; GLdouble:*z*;) – This method sets the value of ***SetAffine*::*scale*** to **::*make_vector***(3, *x*, *y*, *z*).

method:*setScale*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*scale*** to **::*resize_vector***(3, 1.0, *v*).

method:*setTranslation*(public; void; GLdouble:*x*; GLdouble:*y*; GLdouble:*z*;) – This method sets the value of ***SetAffine*::*trans*** to **::*make_vector***(3, *x*, *y*, *z*).

method:*setTranslation*(public; void; *std*::*vector*<GLdouble>:*v*;) – This method sets the value of ***SetAffine*::*trans*** to **::*resize_vector***(3, 0.0, *v*).

B.3.28. message:SetColor

This message is used to set the color of shapes.

Parent Message Construct: **message:*SetVector***

Derived Message Constructs: None

Receiving Processes: **process:*Shape***, **process:*View***

Sending Processes: **process:*Shape***

Methods:

method:*set*(public; void; *std*::*vector*<GLdouble>:*c*;) – Sets the value of the ***vec*** to **::*resize_vector*(**4, 1.0, *c*).

method:*set*(public; void; GLdouble:*r*; GLdouble:*g*; GLdouble:*b*; GLdouble:*a*;) – Sets the value of ***vec*** to ::***make_vector***(4, ***r***, ***g***, ***b***, *a*).

method:*set*(public; void; GLdouble:*r*; GLdouble:*g*; GLdouble:*b*;) – Sets the value of ***vec*** to **::*make_vector***(4, *r*, *g*, *b*, 1.0).

method:*red*(public; GLdouble;) – This routine returns the red color component, ***vec***[0].

method:*green*(public; GLdouble;) – This routine returns the green color component, ***vec***[1].

method:*blue*(public; GLdouble;) – This routine returns the blue color component, ***vec***[2].

method:*alpha*(public; GLdouble;) – This routine returns the alpha color component, ***vec***[1].

B.3.29. message:SetConeSize

This message construct is used to set the parameters of a cone.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*Cone***, **process:*View3D***

Sending Processes: **process:*Cone***

Data Members:

GLdouble:*base* – Specifies the radius of the cone base.

GLdouble:*height* – Specifies the cone height.

GLint:*slices* – Specifies the number of radial slices into which GLUT should break the cone.

GLint:*stacks* – Specifies the number of stacked into which GLUT should break the cone.

Methods:

method:*set*(public; void; GLdouble:*b*; GLdouble:*h*; GLint:*sl*; GLint:*st*;) – Sets ***base*** to *b*, ***height*** to *h*, ***slices*** to *sl*, and ***stacks*** to *st*.

B.3.30. message:SetCubeSize

This message construct is used to set the size of the cube.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*Cube***, **process:*View3D***

Sending Processes: **process:*Cube***

Data Members:

GLdouble:*size* – New edge length for the cube.

Methods:

method:set(public; void; GLdouble:*s*;) – Sets ***size*** to *s*.

B.3.31. message:SetCylinderSize

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*Cylinder***, **process:*View3D***

Sending Processes: **process:*Cylinder***

Data Members:

GLdouble:*radius* – Value for the radius of the cylinder.

GLdouble:*length* – Value for the length of the cylinder.

GLint:*sides* – Number of sides per ring.

GLint:*rings* – Number of radial slices

Methods:

method:*set*(public; void; GLdouble:*i*; GLdouble:*o*; GLint:*s*; GLint:*r*;) – Sets ***inner*** to *i*, ***outer*** to *o*, ***sides*** to *s*, and ***rings*** to *r*.

B.3.32. message:SetDefaultActive

This message is used to set the default value of the active flags within a **process:*Object*** instance. It does not change any of the existing flag values for ***gvm::Object*** instance in any views in which the **process:*Object*** is already registered. Instead, any future registrations will create the ***gvm::Object*** with the default activation flag having been set to the new value specified herein.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*Object***, **process:*View***

Sending Processes: **process:*Object***

Data Members:

bool:*active*(true) – Used to set the default active flag in the receiving **process:*Object*** instance.

Methods:

method:*set*(public; void; bool:*a*;) – Sets ***active*** to *a*.

B.3.33. message:SetLabel

Each ***gvm*::*Object*** can have a label to assist in debugging. This label can be set through the corresponding **process:*Object*** instance in the simulation by first sending a message to that **process:*Object*** and then by forwarding new **message:*SetLabel*** instances to the views where the **process:*Object*** is registered.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*Object*, process:*View***

Sending Processes: **process:*Object***

Data Members:

std*::*string*:*label – The value of the new label for the object.

Methods:

method:*set*(public; void; *std*::*string*:*c*;) – Set ***label*** to *c*.

method:*get*(public; *std*::*string*;) – Return ***label*** to the calling routine.

B.3.34. message:SetMode

Message to set the rendering mode for shapes. The mode can take on any of the OpenGL rendering modes, listed below. One note on this is that the pre-defined 3D shapes (**process:*Cube***, **process:*Cone***, etc) use GLUT to actually perform the rendering. GLUT only allows solid and wire frame rendering modes for these shapes.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*Shape***, **process:*View2D***, **process:*View3D***

Sending Processes: **process:*Shape***

Data Members:

GLenum:*gr_mode* – Takes on one of the following values **GL_POINTS**, **GL_LINES**, **GL_LINE_STRIP**, **GL_LINE_LOOP**, **GL_TRIANGLES**, **GL_TRIANGLE_STRIP**, **GL_TRIANGLE_FAN**, **GL_QUADS**, **GL_QUAD_STRIP**, **GL_POLYGON**. The meaning of these values can be found in (Wright 1996) page 172.

Methods:

method:*set*(public; void; GLenum:*m*;) – Sets ***mode*** to *m*.

B.3.35. message:SetPointSize

This message is sent to a view to set the point size parameter for rendering single vertices in the receiving view. It will cause the **process:*View*** instance to set the ***size*** parameter to the size portion of the payload of the message. When a screen is rendered, the **process:*View*** instance will call ***::glPointSize(size)*** prior to rendering the scene.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*View***

Sending Processes: None

Data Members:

GLfloat:*size*(1.0) – Size parameter for single vertices in the receiving view to be rendered.

Methods:

method:set(**public; void; GLfloat:***s***;**) – Sets size to *s*.

B.3.36. message:SetPosition

Sets location of the GLUT window.

Parent Message Construct: None

Derived Message Constructs: None

Receiving Processes: **process:*View***

Sending Processes: None

Data Members:

int:*x* – X coordinate of the window left side.

int:*y* – Y coordinate of the window top side.

Methods:

method:*set*(public; void; int:*X***; int:***Y***;)** – Sets ***x*** to *X* and ***y*** to *Y*.

B.3.37. message:SetRefresh

Used to set the simulation time between view refreshes.

Parent Message Construct: None

Derived Message Constructs: None

Receiving Processes: **process:*View***

Sending Processes: None

Data Members:

double:*refreshInterval* – Simulation time delta between view refreshes.

Methods:

method:*set*(public; void; double:*r*;) – Sets ***refreshInterval*** to *r*.

B.3.38. message:SetRotation2D

This message is used to set the rotation angle (about the Z axis) for a **process:*Node2D*** instance.

Parent Message Construct: **message:*SetVector2D***

Derived Message Constructs: None

Receiving Processes: **process:*Node2D***, **process:*View2D***

Sending Processes: **process:*Node2D***

B.3.39. message:SetRotation3D

This message is used to set the rotation angles for a **process:*Node3D*** instance.

Parent Message Construct: **message:*Vector3D***

Derived Message Constructs: None

Receiving Processes: **process:*Node3D***, **process:*View3D***

Sending Processes: **process:*Node3D***

B.3.40. message:SetRotationCenter2D

This message is used to set the center of rotation for a **process:*Node2D*** instance.

Parent Message Construct: **message:*SetVector2D***

Derived Message Constructs: None

Receiving Processes: **process:*Node2D***, **process:*View2D***

Sending Processes: **process:*Node2D***

B.3.41. message:SetRotationCenter3D

This message is used to set the center of rotation for a **process:*Node3D*** instance.

Parent Message Construct: **message:*SetVector3D***

Derived Message Constructs: None

Receiving Processes: **process:*Node3D***, **process:*View3D***

Sending Processes: **process:*Node3D***

B.3.42. message:SetScale2D

This message is used to set the scaling factor for a **process:*Node2D*** instance.

Parent Message Construct: **message:*SetVector2D***

Derived Message Constructs: None

Receiving Processes: **process:*Node2D***, **process:*View2D***

Sending Processes: **process:*Node2D***

B.3.43. message:SetScale3D

This message is used to set the scaling factor for a **process:*Node3D*** instance.

Parent Message Construct: **message:*SetVector3D***

Derived Message Constructs: None

Receiving Processes: **process:*Node3D***, **process:*View3D***

Sending Processes: **process:*Node3D***

B.3.44. message:SetScaleCenter2D

This message is used to set the scaling center for a **process:*Node2D*** instance.

Parent Message Construct: **message:*SetVector2D***

Derived Message Constructs: None

Receiving Processes: **process:*Node2D***, **process:*View2D***

Sending Processes: **process:*Node2D***

B.3.45. message:SetScaleCenter3D

This message is used to set the scaling center for a **process:*Node3D*** instance.

Parent Message Construct: **message:*SetVector3D***

Derived Message Constructs: None

Receiving Processes: **process:*Node3D***, **process:*View3D***

Sending Processes: **process:*Node3D***

B.3.46. message:SetSize

This message requests a change in the size of the GLUT view port window.

Parent Message Construct: None

Derived Message Constructs: None

Receiving Processes: **process:*View***

Sending Processes: None

Data Members:

int:*width* – New width of the view port window.

int:*height* – New height of the view port window.

Methods:

method:*set*(public; void; int:*w*; int:*h*;) – Sets ***width*** to *w* and ***height*** to *h*.

B.3.47. message:SetSphereSize

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*Sphere***, **process:*View3D***

Sending Processes: **process:*Sphere***

Data Members:

GLdouble:*radius* – Radius of the sphere.

GLint:*slices* – Number of radial slices in the sphere.

GLint:*stacks* – Number of lateral stacks in the sphere.

Methods:

method:*set*(public; void; GLdouble:*r*; GLint:*sl*; GLint:*st*;) – Sets ***radius*** to *r*, ***slices*** to *sl*, and ***stacks*** to *st*.

B.3.48. message:SetTorusSize

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: None

Receiving Processes: **process:*Torus***, **process:*View3D***

Sending Processes: **process:*Torus***

Data Members:

GLdouble:*inner* – Value for the inner radius of the torus.

GLdouble:*outer* – Value for the outer radius of the torus.

GLint:*sides* – Number of sides per ring.

GLint:*rings* – Number of radial slices

Methods:

method:*set*(public; void; GLdouble:*i*; GLdouble:*o*; GLint:*s*; GLint:*r*;) – Sets ***inner*** to *i*, ***outer*** to *o*, ***sides*** to *s*, and ***rings*** to *r*.

B.3.49. message:SetTranslation2D

This message is used to set the translation for a **process:*Node2D*** instance.

Parent Message Construct: **message:*SetVector2D***

Derived Message Constructs: None

Receiving Processes: **process:*Node2D***, **process:*View2D***

Sending Processes: **process:*Node2D***

B.3.50. message:SetTranslation3D

This message is used to set the translation for a **process:*Node3D*** instance.

Parent Message Construct: **message:*SetVector3D***

Derived Message Constructs: None

Receiving Processes: **process:*Node3D***, **process:*View3D***

Sending Processes: **process:*Node3D***

B.3.51. message:SetValue

Parent Message Construct: None

Derived Message Constructs: **message:*AddView***, **message:*SetAffine***, **message:*SetConeSize***, **message:*SetCubeSize***, **message:*SetDefaultActive***, **message:*SetMode***, **message:*SetSphereSize***, **message:*SetTorusSize*, message:*SetVector***

Receiving Processes: None

Sending Processes: None

Data Members:

***gvm*::*object_index*:*index*((ulong) (-1))** – Index of sending graphics object. Used only when the destination is a View process.

B.3.52. message:SetVector

Base class for messages sending ***std*::*vector*<::GLdouble>** instances between processes.

Parent Message Construct: **message:*SetValue***

Derived Message Constructs: **message:*SetColor***, **message:*SetVector2D***, **message:*SetVector3D***

Receiving Processes: None

Sending Processes: None

Data Members:

::GLdouble:*vec*[] – Vector to send to the destinations.

Methods:

method:*get*(public; *std*::*vector*<::GLdouble>;) – Return ***vec*** to the calling routine.

method:*get*(public; ::GLdouble; ulong:*i*;) – Return the *i*th component of ***vec*** to the calling routine.

B.3.53. message:SetVector2D

Base class for messages sending two dimensional ***std*::*vector*<::GLdouble>** instances between processes.

Parent Message Construct: **message:*SetVector***

Derived Message Constructs: **message:*SetRotation2D***, **message:*SetRotationCenter2D***, **message:*SetScale2D***, **message:*SetScaleCenter2D***, **message:*SetTranslation2D***

Receiving Processes: None

Sending Processes: None

Methods:

method:*init*(public; void;) – Initializes the ***SetVector*::*vec*** to <0.0, 0.0> or, if this is instantiated as a **message:*SetScale2D***, it is initialized to <1.0, 1.0>.

method:*set*(public; void; ::GLdouble:*x*; ::GLdouble:*y*;) – Sets the value of ***SetVector*::*vec*** to **::make_vector(2,** *x*, *y*).

method:*set*(public; void; *std*::*vector*<::GLdouble>:*v*;) – Set the value of ***SetVector*::*vec*** to ::***resize_vector***(2, 0.0, v) or, if the message is instantiated as a **message:*SetScale2D***, it is set to ::***resize_vector***(2, 1.0, *v*).

B.3.54. message:SetVector3D

Base class for messages sending three dimensional ***std*::*vector*<::GLdouble>** instances between processes.

Parent Message Construct: **message:*SetVector***

Derived Message Constructs: **message:*SetRotation3D***, **message:*SetRotationCenter3D***, **message:*SetScale3D***, **message:*SetScaleCenter3D***, **message:*SetTranslation3D***

Receiving Processes: None

Sending Processes: None

Methods:

method:*init*(public; void;) – Initializes the ***SetVector*::*vec*** to <0.0, 0.0, 0.0> or, if this is instantiated as a **message:*SetScale3D***, it is initialized to <1.0, 1.0, 1.0>.

method:*set*(public; void; ::GLdouble:*x*; ::GLdouble:*y*; ::GLdouble:*z*;) – Set the value of ***SetVector*::*vec*** to **::make_vector(3,** *x*, *y*, *z*).

method:*set*(public; void; *std*::*vector*<::GLdouble>:*v*;) – Set the value of ***SetVector*::*vec*** to **::*resize_vector***(3, 0.0, v) or, if the message is instantiated as a **message:*SetScale3D***, it is set to **::*resize_vector***(3, 1.0, *v*).

B.3.55. message:SetVertex2D

A message to set the location of ***a*** **process:*Vertex2D*** instance.

Parent Message Construct: **message:*SetVector2D***

Derived Message Constructs: None

Receiving Processes: **process:*Vertex2D***, **process:*View2D***

Sending Processes: **process:*Vertex2D***

B.3.56. message:SetVertex3D

A message to set the location of a **process:*Vertex3D*** instance.

Parent Message Construct: message:*SetVector3D*

Derived Message Constructs: None

Receiving Processes: **process:*Vertex3D***, **process:*View3D***

Sending Processes: **process:*Vertex3D***

B.3.57. process:Cone

This process construct allows a cone to be viewed within multiple **process:*View3D*** instances. It references a ***gvm*::*Cone*** instance owned by each ***gvm*::*View3D*** instance with which the **process:*Cone*** is registered.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

Data Members:

double:*base*(0.0) – Size of the cone base.

double:*height*(0.0) – Size of the cone base.

int:*slices*(0) – Number of slices into which the cone is segmented.

int:*stacks*(0) – Number of stacks composing the cone.

Methods:

method:*set*(public; void; double:*b*; double:*h*; int:*sl*; int:*st*;) – Sets ***base*** to *b*, ***height*** to *h*, ***slices*** to *sl*, and ***stacks*** to *st*.

mode:*Default* Nodes:

node:*addView*[*AddView*:*in*][*SetConeSize*:*out*]– Upon receiving a request to add view handle, this node will report to the new view the current size parameters of the cone.

node:*setSize*[*SetConeSize*:*in*][*SetConeSize*:*out*[]]– Upon receiving a change in any of the size parameters for the cone, this node will Sets ***base*** to ***in.base***, ***height*** to ***in.height***, ***slices*** to ***in.slices***, and ***stacks*** to ***in.stacks*** and then report these changes to the views in which this cone has been registered. This will allow the associated ***gvm::Cone*** instance in those views to be properly updated.

B.3.58. process:Cube

This process construct allows a cube to be viewed within multiple **process:*View3D*** instances. It references a ***gvm::Cube*** instance owned by each **process:*View3D*** process in which this cube instance has registered.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

Data Members:

double:*size*(0.0) – Length of the cube edges

mode:*Default* Nodes:

node:*addView*[*AddView*:*in*][*SetCubeSize*:*out*] – Upon receiving a request to add a view handle, this node will report to that view the current size parameter of the cube.

node:*setSize*[*SetCubeSize*:*in*][*SetCubeSize*:*out*[]]– Upon receiving a change in any of the size parameter for the cube, this node set ***size*** to ***in.size*** and report to the parent views in which it is registered, those changes. This will allow the associated ***gvm::Cube*** instance in those views to be properly updated.

B.3.59. process:Cylinder

This process construct allows a cylinder to be viewed within multiple **process:*View3D*** instances. It references a ***gvm*::*Cylinder*** instance owned by each **process:*View3D*** process in which this process is registered.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

Data Members:

double:*radius*(0.0) – Radius of the cylinder.

double:*length*(0.0) – Length of the cylinder.

int:*sides*(0) – Number of sides the cylinder will have.

int:*rings*(0) – Number of rings the cylinder will have.

Methods:

method:*set*(public; void; double:*rad*; double:*l*; int:*s*; int:*r*;) – Sets ***radius*** to *rad*, ***length*** to *l*, ***sides*** to *s*, ***rings*** to *r*.

mode:*Default* Nodes:

node:*addView*[*AddView*:*in*][*SetCylinderSize*:*out*] – Upon receiving a request to add a view handle, this node will report to that view the current size parameters of the cylinder.

node:*setSize*[*SetCylinderSize*:*in*][*SetCylinderSize*:*out*[]] – Upon receiving a change in any of the parameters for the cylinder, this node will report to the **process:*View*** instances where this cylinder is registered to update the associated ***gvm*::*Cylinder*** instances.

B.3.60. process:Dodecahedron

This process construct allows a dodecahedron to be viewed within multiple **process:*View3D*** instances. It references a ***gvm*::*Dodecahedron*** instance owned by each **process:*View3D*** process in which this process is registered.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

B.3.61. process:Icosahedron

This process construct allows a dodecahedron to be viewed within multiple **process:*View3D*** instances. It references a ***gvm*::*Icosahedron*** instance owned by each **process:*View3D*** process in which this process is registered.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

B.3.62. process:Node

Nodes server the purpose of retaining a list of subordinate **process:*Node*** and **process:*Shape*** instances and the affine transformation information for both. They have an associate ***gvm*::*Node*** instance directly managed in the view processes in which this node is registered.

Parent Process Construct: **process:*Object***

Derived Process Constructs: **process:*Node2D***, **process:*Node3D***

Member Data:

GLdouble:*color*[4](-1.0) – Default color for all subordinate processes. The default value (-1.0) indicates to the associate ***gvm*::*Node*** instances that they are to derive their color from parent ***gvm*::*Node*** instances.

process:*nodeList*[] – List of subordinate **process:*Node*** instances.

GLfloat:*ptSize*(-1.0) – Default point size for all subordinate processes. The default value (-1.0) indicates to the associate ***gvm*::*Node*** instances that they are to derive their point size from parent ***gvm*::*Node*** instances.

process:*shapeList*[] – List of subordinate **process:*Shape*** instances.

Methods:

method:*set_color*(public; void; GLdouble:*red*; GLdouble:*green*; GLdouble:*blue*; GLdouble:*alpha*;) – Set ***color*** to <*red*, *green*, *blue*, *alpha*>

method:*set_color*(public; void; *std*::*vector*<GLdouble>:*c*;) – Set ***color*** to *c*.

method:*set_point_size*(public; void; GLfloat:*ps*;) – Set ***ptSize*** to *ps*.

mode:*Default* nodes:

node:*addView*[*AddView*:*in*][*SetColor*:*sc*, *SetPointSize*:*sps*] – Upon notification of the addition of this **process:*Node*** instance to a ***gvm*::*View*** associated with a **process:*View***, this node will set the default color and point size settings for the associate ***gvm*::*Node***.

node:*setColor*[*SetColor*:*in*][*SetColor*:*out*[]]– Upon a color change request, all of the **process:*View*** instances with ***gvm*::*Node*** instances associated with this **process:*Node*** are notified of the change.

node:*setPointSize*[*SetPointSize*:*in*][*SetPointSize*:*out*[]]– Upon a point size change request, all of the **process:*View*** instances with ***gvm*::*Node*** instances associated with this **process:*Node*** are notified of the change.

B.3.63. process:Node2D

This process construct further specializes **process:*Node*** to govern two-dimensional graphics objects.

Parent Process Construct: **process:*Node***

Derived Process Constructs: None

Data Members:

double:*ctrRot*[2](0.0) – Center of rotation.

double:*ctrScale*[2](0.0) – Center for the scaling.

double:*rot*[2](0.0) – Rotation angles.

double:*scale*[2](1.0) – Scaling factors.

double:*trans*[2](0.0) – Translation factors.

mode:*Default* Nodes:

node:*addNode*[*AddNode2D*:*in*][*RegisterNode2D*:*out*[]] – This will add subnodes to this **process:*Node2D***. The new subnodes will then need to be registered with all **process:*View2D*** instances with which this **process:*Node2D*** is registered. Sending a message:RegisterNode2D message to those process:View2D instances does this.

node:*addShape*[*AddShape2D*:*in*][*RegisterShape2D*:*out*[]] – This will add subnodes to this **process:*Shape2D***. The new shapes will then need to be registered with the **process:*View2D*** instances with which this **process:*Node2D*** is registered. Sending a message:RegisterShape2D message to those process:View2D instances does this.

node:*addView*[*AddView*:*in*][*RegisterNode2D*:*rn*, *RegisterShape2D*:*rs*, *SetAffine2D*:*sa*] – Upon receipt of a **message:*AddView*** instance, this **process:*Node2D*** needs to register all of its subnodes and shapes with that view. The **message:*RegisterNode2D*** and **message:*RegisterShape2D*** messages do that. In addition, this **process:*Node2D*** instance needs to inform the **process:*View2D*** as to the affine transform parameters.

node:*setAffine*[*SetAffine2D*:*in*][*SetAffine2D*:*out*[]] – This node sets the affine transform parameters of this **process:*Node2D*** instance. That change is then passed to the **process:*View2D*** instances in which this **process:*Node2D*** is registered in order to update the associate ***gvm*::*Node2D*** affine transform parameters.

node:*setCtrRot*[*SetRotationCenter2D*:*in*][*SetRotationCenter2D*:*out*[]] – This message sets the rotation center of this **process:*Node2D*** instance. That change is then passed to the **process:*View2D*** instances in which this **process:*Node2D*** is registered to update the associate ***gvm*::*Node2D*** rotation center.

node:*setCtrScale*[*SetScaleCenter2D*:*in*][*SetScaleCenter2D*:*out*[]] – This message sets the scaling center of this **process:*Node2D*** instance. That change is then passed to the **process:*View2D*** instances in which this **process:*Node2D*** is registered to update the associate ***gvm*::*Node2D*** scaling center.

node:*setRotation*[*SetRotation2D*:*in*][*SetRotation2D*:*out*[]] – This message sets the rotation angle (in radians) of this **process:*Node2D*** instance. That change is then passed to the **process:*View2D*** instances in which this **process:*Node2D*** is registered to update the associate ***gvm*::*Node2D*** rotation value.

node:*setScale*[*SetScale2D*:*in*][*SetScale2D*:*out*[]] – This message sets the scale of this **process:*Node2D*** instance. That change is then passed to the **process:*View2D*** instances where this **process:*Node2D*** is registered so that the associated ***gvm*::*Node2D*** instances may have their scaling factors updated.

node:*setTranslation*[*SetTranslation2D*:*in*][*SetTranslation2D*:*out*[]] – This message sets the translation factor of this **process:*Node2D*** instance. That change is then passed to the **process:*View2D*** instances in which this **process:*Node2D*** is registered to update the associate ***gvm*::*Node2D*** translation factors.

B.3.64. process:Node3D

This process construct further specializes **process:*Node*** to govern two-dimensional graphics objects.

Parent Process Construct: **process:*Node***

Derived Process Constructs: None

Data Members:

double:*ctrRot*[3](0.0) – Center of rotation.

double:*ctrScale*[3](0.0) – Center for the scaling.

double:*rot*[3](0.0) – Rotation angles.

double:*scale*[3](1.0) – Scaling factors.

double:*trans*[3](0.0) – Translation factors.

mode:*Default* **Nodes**:

node:*addNode*[*AddNode3D*:*in*][*RegisterNode3D*:*out*[]] – This will add subnodes to this **process:***Node3D*. The new subnodes will then need to be registered with all **process:***View3D* instances with which this **process:***Node3D* is registered. Sending a message:RegisterNode3D message to those process:View3D instances does this.

node:*addShape*[*AddShape3D*:*in*][*RegisterShape3D*:*out*[]] – This will add subnodes to this **process:***Shape3D*. The new shapes will then need to be registered with the **process:***View3D* instances with which this **process:***Node3D* is registered. Sending a message:RegisterShape3D message to those process:View3D instances does this.

node:*addView*[*AddView*:*in*][*RegisterNode3D*:*rn*, *RegisterShape3D*:*rs*, *SetAffine3D*:*sa*] – Upon receipt of a **message:***AddView* instance, this **process:***Node3D* needs to register all of its subnodes and shapes with that view. The **message:***RegisterNode3D* and **message:***RegisterShape3D* messages do that. In addition, this **process:***Node3D* instance needs to inform the **process:***View3D* as to the affine transform parameters.

node:*setAffine*[*SetAffine3D*:*in*][*SetAffine3D*:*out*[]] – This message sets the affine transform parameters of this **process:***Node3D* instance. That change is then passed to the **process:***View3D* instances in which this **process:***Node3D* is registered in order to update the associate ***gvm*::*Node3D*** affine transform parameters.

node:*setCtrRot*[*SetRotationCenter3D*:*in*][*SetRotationCenter3D*:*out*[]] – This message sets the rotation center of this **process:***Node3D* instance. That change is then passed to the **process:***View3D* instances in which this **process:***Node3D* is registered to update the associate ***gvm*::*Node3D*** rotation center.

node:*setCtrScale*[*SetScaleCenter3D*:*in*][*SetScaleCenter3D*:*out*[]] – This message sets the scaling center of this **process:*Node3D*** instance. That change is then passed to the **process:*View3D*** instances in which this **process:*Node3D*** is registered to update the associate ***gvm*::*Node3D*** scaling center.

node:*setRotation*[*SetRotation3D*:*in*][*SetRotation3D*:*out*[]] – This message sets the rotation angle (in radians) of this **process:*Node3D*** instance. That change is then passed to the **process:*View3D*** instances in which this **process:*Node3D*** is registered to update the associate ***gvm*::*Node3D*** rotation value.

node:*setScale*[*SetScale3D*:*in*][*SetScale3D*:*out*[]] – This message sets the scale of this **process:*Node3D*** instance. That change is then passed to the **process:*View3D*** instances where this **process:*Node3D*** is registered so that the associated ***gvm*::*Node3D*** instances may have their scaling factors updated.

node:*setTranslation*[*SetTranslation3D*:*in*][*SetTranslation3D*:*out*[]] – This message sets the translation factor of this **process:*Node3D*** instance. That change is then passed to the **process:*View3D*** instances in which this **process:*Node3D*** is registered to update the associate ***gvm*::*Node3D*** translation factors.

B.3.65. process:Object

This process acts as a base class for various GUI process. Each GUI process can be registered in a variety of **process:*View*** instances. Each **process:*View*** provides the **process:*Object*** instance with an index value that messages to each of those views uses to access the associated ***gvm*::*Object*** instance. Additionally, each **process:*Object*** instance can turn itself on and off in each of the views. There is a framework allowing all ***gvm*::*Object*** instances associated with this **process:*Object*** instance to be independently activated.

Parent Process Construct: None

Derived Process Constructs: **process:*Node***, **process:*Shape***, **process:*Vertex***

Data Members:

bool:*defActive*(true) – Default active value.

std*::*map*<process, *gvm*::*object_index*>:*indexMap – Map of object/view indices associations.

mode:*Default* Nodes:

node:*addView*[*AddView*:*in*][*SetActive*:*out*, *SetLabel*:*sl*] – When a new **process:*View*** is added, add that view and its associated index to ***indexMap*** by setting ***indexMap***[***in.view***] to ***in.index***. The **process:*View*** instance is also informed as to the default active status and the process label.

node:*selectiveActivate*[*SelectiveActivate*:*in*][*SetActive*:*out*[]] – Here, we can selectively activate or deactivate the associated ***gvm*::*Object*** instances according to the associate made in ***in.activeMap***. Those affected **process:*View*** instances are notified of the change with the outbound **message:*SetActive***.

node:*setActive*[*SetActive*:*in*][*SetActive*:*out*[]] – This sends **message:*SetActive*** instances to all views in ***indexMap*** reflecting the change in the active status. The result is that all active flags in the associated ***gvm*::*Object*** instances in all views in which this object is registered are set to ***in.active*** (i.e. they can all be turned on or off with this one inbound message).

node:*setDefaultActive*[*SetDefaultActive*:*in*][] – This sets the ***defActive*** flag to the ***in.active*** flag. It only sets the active flag, and does not change the view/activity state associations.

node:*setLabel*[*SetLabel*:*in*][*SetLabel*:*out*[]] – This sets the ***defActive*** flag to the ***in.active*** flag. It only sets the active flag, and does not change the view/activity state associations.

B.3.66. process:Octahedron

This process construct allows a octahedron to be viewed within multiple **process:*View3D*** instances. It references a ***gvm*::*Octahedron*** instance owned by each **process:*View3D*** process in which this process is registered.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

B.3.67. process:Polygon2D

This process construct is for general two-dimensional polygons. The rendering mode for this polygon and the location of the polygon vertices must be consistent with the restrictions provided within OpenGL. The most prominent of these restrictions is that if the mode value is **GL_POLYGON**, then vertices must form a convex polygon.

Parent Process Construct: **process:*Shape2D***

Derived Process Constructs: None

Data Members:

process:*vertexList*[] – List of vertices for this polygon. They will be rendered in the order added to the list.

Methods:

mode:*Default* **Nodes**:

node:*addVertex*[*AddVertex2D*:*in*][*RegisterVertex2D*:*out*[]] – When new vertices are added, these additions need to be registered with the **process:*View2D*** instance that manages the associate ***gvm*::*Polygon2D*** instances.

node:*addView*[*AddView*:*in*][*RegisterVertex2D*:*rv*] – Upon receipt of the **message:*AddView***, this **process:*Polygon2D*** will register the vertices comprising it with the view that sent the **message:*AddView***.

B.3.68. process:Polygon3D

This process construct is for general three-dimensional polygons. The rendering mode for this polygon and the location of the polygon vertices must be consistent with the restrictions provided within OpenGL. The most prominent of these restrictions is that if the mode value is **GL_POLYGON**, then vertices must form a convex polygon.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

Data Members:

process:*vertexList*[] – List of vertices for this polygon. They will be rendered in the order added to the list.

Methods:

mode:*Default* Nodes:

node:*addVertex*[*AddVertex3D*:*in*][*RegisterVertex3D*:*out*[]] – When new vertices are added, these additions need to be registered with the **process:*View3D*** instances that manage the associate ***gvm*::*Polygon3D*** instances.

node:*addView*[*AddView*:*in*][*RegisterVertex3D*:*rv*] – Upon receipt of the **message:*AddView***, this **process:*Polygon3D*** will register the vertices comprising it with the view that sent the **message:*AddView***.

B.3.69. process:Shape

This is the parent construct for all of the shapes. Shapes have applied to them affine transforms to position them somewhere in the scene. This is then viewed from a certain position.

Parent Process Construct: **process:*Object***

Derived Process Constructs: **process:*Shape2D***, **process:*Shape3D***

Data Members:

GLdouble:*color*[4](-1.0) – This is the color associated with this shape instance. The default value (-1.0) indicates to the associate ***gvm*::*Shape*** instances that they are to derive their color from parent ***gvm*::*Node*** instances.

GLenum:*gr_mode* – Rendering mode for this shape. It takes on one of the following values **GL_POINTS**, **GL_LINES**, **GL_LINE_STRIP**, **GL_LINE_LOOP**, **GL_TRIANGLES**,

GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, GL_POLYGON. The meaning of these values can be found in (Wright 1996) page 172.

GLfloat:*ptSize*(-1.0) – Point size attribute of this shape. The default value (-1.0) indicates to the associate ***gvm*::*Shape*** instances that they are to derive their point size attribute from parent ***gvm*::*Node*** instances.

Methods:

method:*set_color*(public; void; GLdouble:*r*; GLdouble:*g*; GLdouble:*b*; GLdouble:*a*;) – Sets ***color*** to ::***make_vector***(4, *r*, *g*, *b*, *a*).

method:*set_color*(public; void; *std*::*vector*<GLdouble>:*c*;) – Sets ***color*** to ::***resize_vector***(4, 1.0, *c*).

method:*set_mode*(public; void; GLenum:*m*;) – Sets ***gr_mode*** to *m*.

method:*set_point_size*(public; void; GLfloat:*ps*;) – Sets ***pt_size*** to *ps*.

mode:*Default* Nodes:

node:*addView*[*AddView*:*in*][*SetColor*:*sc*, *SetMode*:*sm*, *SetPointSize*:*sps*] – Upon **message:*AddView***, this process will inform the new **process:*View*** instance of the shape's color, rendering mode, and point size.

node:*setColor*[*SetColor*:*in*][*SetColor*:*out*[]] – When modifying the color of the shape, the change is passed to the **process:*View*** instances where this shape is registered so that the associate ***gvm*::*Shape*** instances can have their color parameters similarly altered.

node:*setMode*[*SetMode*:*in*][*SetMode*:*out*[]] – When modifying the rendering mode of the shape, the change is passed to the **process:*View*** instances where this shape is registered so that the associate ***gvm*::*Shape*** instances can have their rendering mode parameters similarly altered.

node:*setPointSize*[*SetPointSize*:*in*][*SetPointSize*:*out*[]] – When modifying the point size of the shape, the change is passed to the **process:*View*** instances where this shape is registered so that the associate ***gvm*::*Shape*** instances can have their point size parameters similarly altered.

B.3.70. process:Shape2D

This further specializes the **process:*Shape*** construct to allow two dimensional shapes within a **process:*View2D*** instance.

Parent Process Construct: **process:*Shape***

Derived Process Constructs: **process:*Polygon2D***

B.3.71. process:Shape3D

This further specializes the **process:*Shape*** construct to allow three dimensional shapes within a **process:*View3D*** instance.

Parent Process Construct: **process:*Shape***

Derived Process Constructs: **process:*Cone***, **process:*Cube***, **process:*Cylinder***, **process:*Dodecahedron***, **process:*Icosahedron***, **process:*Octahedron***, **process:*Sphere***, **process:*Tetrahedron***

B.3.72. process:Sphere

This process construct allows a sphere to be viewed within multiple **process:*View3D*** instances. It references a ***gvm::Sphere*** instance owned by each **process:*View3D*** process in which this process is registered.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

Data Members:

double:*radius*(0.0) – Sphere radius.

int:*slices*(0) – Number of slices through the sphere.

int:*stacks*(0) – Number of stacks in the sphere.

Methods:

method:set(public; void; double:*r***; int:***sl***; int:***st***;)** – Sets ***radius*** to *r*, ***slices*** to *sl*, and ***stacks*** to *st*.

mode:*Default* Nodes:

node:*addView*[*AddView*:*in*][*SetSphereSize*:*out*] – Upon receiving a request to add a new **process:*View*** handle, this node will report to that view the current size parameters of the sphere.

node:*setSize*[*SetSphereSize*:*in*][*SetSphereSize*:*out*[]] – Upon receiving a change in any of the parameters for the sphere, this node will report to the **process:*View*** instances where this **process:*Sphere*** is registered to update the associated ***gvm::Sphere*** instances.

B.3.73. process:Tetrahedron

This process construct allows a tetrahedron to be viewed within multiple **process:*View3D*** instances. It references a ***gvm::Tetrahedron*** instance owned by each **process:*View3D*** process in which this process is registered.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

B.3.74. process:Torus

This process construct allows a torus to be viewed within multiple **process:*View3D*** instances. It references a ***gvm::Torus*** instance owned by each **process:*View3D*** process in which this process is registered.

Parent Process Construct: **process:*Shape3D***

Derived Process Constructs: None

Data Members:

double:*inner*(0.0) – Size of the inner radius.

double:*outer*(0.0) – Size of the outer radius.

int:*sides*(0) – Number of sides the torus will have.

int:*rings*(0) – Number of rings the torus will have.

Methods:

method:*set*(public; void; double:*i*; double:*o*; int:*s*; int:*r*;) – Sets ***inner*** to *i*, ***outer*** to *o*, ***sides*** to *s*, ***rings*** to *r*.

mode:*Default* Nodes:

node:*addView*[*AddView*:*in*][*SetTorusSize*:*out*] – Upon receiving a request to add a view handle, this node will report to that view the current size parameters of the torus.

node:*setSize*[*SetTorusSize*:*in*][*SetTorusSize*:*out*[]] – Upon receiving a change in any of the parameters for the torus, this node will report to the **process:*View*** instances where this torus is registered to update the associated ***gvm*::*Torus*** instances.

B.3.75. process:Vertex

Parent Process Construct: **process:*Object***

Derived Process Constructs: **process:*Vertex2D***, **process:*Vertex3D***

B.3.76. process:Vertex2D

This is a two-dimensional vertex. It is normally subordinated to a **process:*Polygon2D***. It is associated with a ***gvm*::*Vertex2D*** which is manages in the parent **process:*View2D***.

Parent Process Construct: **process:*Vertex***

Derived Process Constructs: None

double:*vert*[2](0.0) – Vertex location, initialized to <0.0, 0.0>

mode:*Default* Nodes:

node:*addView*[*AddView*:*in*][*SetVertex2D*:*out*] – reports to the **process:*Views*** where this vertex is registered to update the associated ***gvm*::*Vertex2D*** instances.

node:*setVertex*[*SetVertex2D*:*in*][*SetVertex2D*:*out*[]] – Sets the vertex location, and reports that new location to the **process:*View2D*** instances where this vertex is registered so that the associated ***gvm*::*Vertex2D*** can properly represent the current state of this vertex.

B.3.77. process:Vertex3D

This is a three-dimensional vertex. It is subordinated to a **process:*Polygon3D***. It is associated with a ***gvm*::*Vertex3D*** which is manages in the parent **process:*View3D***.

Parent Process Construct: **process:*Vertex***

Derived Process Constructs: None

Data Members:

double:*vert*[3](0.0) – Vertex location, initialized to <0.0, 0.0, 0.0>

Methods:

mode:*Default* Nodes:

node:*addView*[*AddView*:*in*][*SetVertex3D*:*out*] – reports to the **process:*Views*** instances where this vertex is registered to update the associated ***gvm*::*Vertex3D*** instances.

node:*setVertex*[*SetVertex3D*:*in*][*SetVertex3D*:*out*[]] – Sets the vertex location, and reports that new location to the new **process:*View3D*** instances where this vertex is registered so that the associated ***gvm*::*Vertex3D*** can properly represent the current state of this vertex.

B.3.78. process:View

A **process:*View*** construct instance manages a single GLUT window in which can be displayed information in any form the programmer wishes.

Parent Process Construct: None

Derived Process Constructs: **process:*View2D***, **process:*View3D***

Data Members:

std*::*vector<std*::*vector<gvm*::*object_index>* >:*procList – Indices associated with specific processes in the simulation system are stored in this structure. The index for some process instance *v* is stored in ***procList***[*v*.***getNode***()][*v*.***getIndex***()]. As new processes request management from a **process:*View*** instance, ***procList*** is polled to determine whether the process has already been registered with the view. If not, ***procList*** is resized if necessary and a unique identifier is placed into the appropriate place in ***procList***.

double:*refreshInterval*(0.01) – Time between consecutive refreshes.

***gvm*::*View**:*view*(NULL)** – This is the actual view that renders the scene.

Methods:

method:*init*(public; void;) – Registers view with the local ***sodl*::*GLUTViewManager*** instance.

method:*get*(protected; *gvm*::*object_index*; process:*p*;) – Returns the object index associated with the process instance with handle *p*. If necessary, this routine will increase the size of the ***procList*** structure. This is

method:*set*(protected; void; process:*p*; *gvm*::*object_index*:*v*;) – This method sets the object instance for process *p* to *v*. ***procList*** is resized to accommodate the new process if necessary.

method:*restore*(public; void;) – This method is called when a rollback restores the state instance receiving the call. In this case, it will call the ***(*view).restore(getTime())*** to remove any pending events in ***view*** that have timestamps later than ***sodl::TimeStamp::getTime()***.

method:*fossilCollect*(public; void;) – This method is called when a fossil collection event occurs for the view. In this case, it will call the ***(*view).fossilCollect(getTime())*** to process any pending events in ***view*** that have timestamps with time stamp ***sodl::TimeStamp::getTime()***.

mode:*Default* Nodes:

node:*refresh*[*RefreshDisplay*:*in*][*RefreshDisplay*:*out*] – When **message:*RefreshDisplay*** instances are received, the node will schedule a ***gvm::Refresh*** event with ***view*** a request to update the display for time stamp ***sodl::TimeStamp::getTime()***. This event will actually be processed during the fossil collection process. The node then sends an output message to schedule another **message:*RefreshDisplay*** event for ***sodl::TimeStamp::getTime()+refreshInterval***.

node:*setActive*[*SetActive*:*in*][] – This node will schedule ***gvm::SetActive*** event to set the active parameter of the ***gvm::Object*** instance with index ***in.index*** to ***in.active*** for time stamp ***sodl::TimeStamp::getTime()***.

node:*setColor*[*SetColor*:*in*][] – This node will schedule a ***gvm::SetColor*** event to set the color parameter of the ***gvm::Object*** instance with index ***in.index*** to ***in.color*** for time stamp ***sodl::TimeStamp::getTime()***.

node:*setLabel*[*SetLabel*:*in*][] – This node will schedule a ***gvm::SetLabel*** event to set the label string parameter of the ***gvm::Object*** instance with index ***in.index*** to ***in.label*** for time stamp ***sodl::TimeStamp::getTime()***.

node:*setMode*[*SetMode*:*in*][] – This node will schedule a ***gvm::SetMode*** event to set the rendering mode parameter of the ***gvm::Object*** instance with index ***in.index*** to ***in.gr_mode*** for time stamp ***sodl::TimeStamp::getTime()***.

node:*setPointSize*[*SetPointSize*:*in*][] – If ***in.index*** references an object, then this node will schedule a ***gvm::SetPointSize*** event in ***view*** to change the point size parameter of the referenced ***gvm::Object*** instance.

If the ***in.index*** does not refer to an object, then it applies to the default point size value for ***view***. In that case, an event for changing ***view***'s point size is scheduled in ***view***. In both cases, the time stamp for these scheduled events is ***sodl*::*TimeStamp*::*getTime***().

node:*setPosition*[*SetPosition*:*in*][] – This node will schedule ***gvm*::*SetPosition*** event with ***view*** to set the position of the GLUT window ***view*** controls to <*in.x*, *in.y*> with time stamp ***sodl*::*TimeStamp*::*getTime***().

node:*setRefreshInterval*[*SetRefresh*:*in*][] – This node will set ***refreshInterval*** to the value provided in ***in.refreshInterval***.

node:*setSize*[*SetSize*:*in*][] – This node will schedule a ***gvm*::*SetSize*** event with ***view*** to set the size of the GLUT window ***view*** controls to <***in.width***, ***in.height***> with time stamp ***sodl*::*TimeStamp*::*getTime***().

node:*start*[*StartSimulation*:*in*][*RefreshDisplay*:*out*] – This node schedules the first refresh event to occur at time 0.0. All subsequent refresh events will occur at intervals of ***refreshInterval***.

B.3.79. process:View2D

The **process:*View2D*** construct provides more specialized user interactions for two-dimensional data representation.

Parent Process Construct: **process:*View***

Derived Process Constructs: None

Methods:

method:*init*(public; void;) – This routine will allocate ***view*** as a ***gvm*::*View2D*** instance and then call ***Node*::*init***() to register that ***gvm*::*View*** instance with the local instance of the ***sodl*::*GLUTViewManager*** instance.

method:*getGVMType*(protected; *gvm*::*object_type*; *sodl*::ptype:*t*;) – This returns the ***gvm*::*object_type*** associated with processes of type *t*.

mode:*Default* Nodes:

node:*addNode*[*AddNode2D*:*in*][*AddView*:*out*[]] – Adds a collection of **process:*Node2D*** handles to this **process:*View2D***, each of which is directly subordinated to ***view***. It does this by scheduling a ***gvm*::*CreateObject*** to request creation of new ***gvm*::*Node2D*** instances in ***view*** for each node *in*.***subordinates*** not previously registered. It then schedules new ***gvm*::*AddNode*** events with the ***view*** for actually subordinating all of the nodes in *in*.***subordinates*** to ***view***. These events are each time stamped for ***sodl*::*TimeStamp*::*getTime*()**. Any new nodes are informed as to their new index value.

node:*regNode*[*RegisterNode2D*:*in*][*AddView*:*out*[]] – This will register a collection of **process:*Node2D*** instances and specify their parent **process:*Node2D*** instance (which is the source of the message *in*). It does this by scheduling a ***gvm*::*CreateObject*** event to request creation of a new ***gvm*::*Node2D*** instances in ***view*** for each node *in*.***subordinates*** not previously registered. A ***gvm*::*AddNode*** message is then scheduled for each of the processes listed in *in*.***subordinates***[], which are added as sub-nodes to the ***gvm*::*Node*** instance with index *in*.***index***. All of these events are each time stamped for ***sodl*::*TimeStamp*::*getTime*()**. Any new nodes are informed as to their new index value.

node:*regShape*[*RegisterShape2D*:*in*][*AddView*:*out*[]] – This will register a collection of **process:*Shape2D*** instances and specify their parent **process:*Node2D*** instance (which is the source of the message *in*). It does this by scheduling a ***gvm*::*CreateObject*** event to request creation of a new ***gvm*::*Shape2D*** instances in ***view*** for each node *in*.***subordinates*** not previously registered. A ***gvm*::*AddShape*** message is then scheduled for each of the processes listed in *in*.***subordinates***[], which are added as subordinate shapes to the ***gvm*::*Node*** instance with index *in*.***index***. All of these events are each time stamped for ***sodl*::*TimeStamp*::*getTime*()**. Any new nodes are informed as to their new index value.

node:*regVertex*[*RegisterVertex2D*:*in*][*AddView*:*out*[]] – This will register a collection of **process:*Vertex2D*** instances and specify their parent **process:*Polygon2D*** instance (which is the source of the message *in*). It does this by scheduling a ***gvm*::*CreateObject*** event to request creation of a new ***gvm*::*Vertex2D*** instances in ***view*** for each node *in*.***subordinates*** not previously registered. A ***gvm*::*AddVertex*** message is then scheduled for each of the processes listed in *in*.***subordinates***[], which are added as subordinate shapes to the ***gvm*::*Polygon2D*** instance with index *in.**index***. All of these events are each time stamped for ***sodl*::*TimeStamp*::*getTime*()**. Any new nodes are informed as to their new index value.

node:*setAffine*[*SetAffine2D*:*in*][] – This node schedules a ***gvm*::*SetAffine*** event with ***view*** an update for all of the affine transformation components in the ***gvm*::*Node2D*** instance with index *in.**index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setRotation*[*SetRotation2D*:*in*][] – This node schedules a ***gvm*::*SetRotation*** event with ***view*** an update for the rotation portion of the affine transformation in the ***gvm*::*Node2D*** instance with index *in.**index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setRotationCenter*[*SetRotationCenter2D*:*in*][] – This node schedules a ***gvm*::*SetRotationCenter*** event with ***view*** an update for the rotational center portion of the affine transformation in the ***gvm*::*Node2D*** instance with index *in.**index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setScale*[*SetScale2D*:*in*][] – This node schedules a ***gvm*::*SetScale*** event with ***view*** an update for the scale portion of the affine transformation in the ***gvm*::*Node2D*** instance with index *in.**index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setScaleCenter*[*SetScaleCenter2D*:*in*][] – This node schedules a ***gvm*::*SetScaleCetner*** event with ***view*** an update for the scaling center portion of the affine transformation in the ***gvm*::*Node2D*** instance with index *in.**index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setTranslation*[*SetTranslation2D*:*in*][] – This node schedules a ***gvm*::*SetTranslation*** event with ***view*** an update for the translation portion of the affine transformation in the ***gvm*::*Node2D*** instance with index *in.**index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setVertex*[*SetVertex2D*:*in*][] – This node will schedule a ***gvm*::*SetVertex*** event with ***view*** a change to the ***gvm*::*Vertex2D*** instance with index *in.**index*** to the vertex value *in.get*(). The time stamp for the scheduled event will be ***sodl*::*TimeStamp*::*getTime*()**.

B.3.80. process:View3D

The **process:*View3D*** construct provides more specialized user interactions for three-dimensional data representation.

Parent Process Construct: **process:*View***

Derived Process Constructs: None

Methods:

method:*init*(public; void;) – This routine will allocate ***view*** as a ***gvm*::*View3D*** instance and then call ***Node*::*init*()** to register that ***gvm*::*View*** instance with the local instance of the ***sodl*::*GLUTViewManager*** instance.

method:*getGVMType*(protected; *gvm*::*object_type*; *sodl*::ptype:*t*;) – This returns the ***gvm*::*object_type*** associated with processes of type ***t***.

mode:*Default* Nodes:

node:*addNode*[*AddNode3D*:*in*][*AddView*:*out*[]] – Adds a collection of **process:*Node3D*** handles to this **process:*View3D***, each of which is directly subordinated to ***view***. It does this by scheduling a ***gvm*::*CreateObject*** to request creation of new ***gvm*::*Node3D*** instances in ***view*** for each node ***in.subordinates*** not previously registered. It then schedules new ***gvm*::*AddNode*** events with the ***view*** for actually subordinating all of the nodes in ***in.subordinates*** to ***view***. These events are each time stamped for ***sodl*::*TimeStamp*::*getTime*()**. Any new nodes are informed as to their new index value.

node:*regNode*[*RegisterNode3D*:*in*][*AddView*:*out*[]] – This will register a collection of **process:*Node3D*** instances and specify their parent **process:*Node3D*** instance (which is the source of the message ***in***). It does this by scheduling a ***gvm*::*CreateObject*** event to request creation of a new ***gvm*::*Node3D*** instances in ***view*** for each node ***in.subordinates*** not previously registered. A ***gvm*::*AddNode*** message is then scheduled for each of the processes listed in ***in.subordinates*[]**, which are added as sub-nodes to the ***gvm*::*Node***

instance with index *in.index*. All of these events are each time stamped for ***sodl::TimeStamp::getTime()***. Any new nodes are informed as to their new index value.

node:*regShape*[*RegisterShape3D*:*in*][*AddView*:*out*[]] – This will register a collection of **process:*Shape3D*** instances and specify their parent **process:*Node3D*** instance (which is the source of the message *in*). It does this by scheduling a ***gvm::CreateObject*** event to request creation of a new ***gvm::Shape3D*** instances in ***view*** for each node *in.**subordinates*** not previously registered. A ***gvm::AddShape*** message is then scheduled for each of the processes listed in *in.**subordinates***[], which are added as sub-nodes to the ***gvm::Node*** instance with index *in.**index***. All of these events are each time stamped for ***sodl::TimeStamp::getTime()***. Any new nodes are informed as to their new index value.

node:*regVertex*[*RegisterVertex3D*:*in*][*AddView*:*out*[]] – This will register a collection of **process:*Vertex3D*** instances and specify their parent **process:*Polygon3D*** instance (which is the source of the message *in*). It does this by scheduling a ***gvm::CreateObject*** event to request creation of a new ***gvm::Vertex3D*** instances in ***view*** for each node *in.**subordinates*** not previously registered. A ***gvm::AddNode*** message is then scheduled for each of the processes listed in *in.**subordinates***[], which are added as sub-nodes to the ***gvm::Node*** instance with index *in.**index***. All of these events are each time stamped for ***sodl::TimeStamp::getTime()***. Any new nodes are informed as to their new index value.

node:*setAffine*[*SetAffine3D*:*in*][] – This node schedules a ***gvm::SetAffine*** event with ***view*** an update for all of the affine transformation components in the ***gvm::Node3D*** instance with index *in.**index***. The time stamp for this scheduled item is ***sodl::TimeStamp::getTime()***.

node:*setConeSize*[*SetConeSize*:*in*][] – This node schedules a ***gvm::SetConeSize*** event with view an event with time stamp ***sodl::TimeStamp::getTime()*** to change the parameters of the ***gvm::Cone*** instance with identifier *in.**index*** to the parameters specified in ***in***.

node:*setCubeSize*[*SetCubeSize*:*in*][] – This node schedules a ***gvm::SetSetCubeSize*** event with view an event with time stamp ***sodl::TimeStamp::getTime()*** to change the parameters of the ***gvm::Cube*** instance with identifier *in.**index*** to the parameters specified in *in*.

node:*setCylinderSize*[*SetCylinderSize*:*in*][] – This schedules a ***gvm*::*SetSetCylinderSize*** event with view an event with time stamp ***sodl*::*TimeStamp*::*getTime*()** to change the parameters of the ***gvm*::*Cylinder*** instance with identifier ***in.index*** to the parameters specified in *in*.

node:*setRotation*[*SetRotation3D*:*in*][] – This node schedules a ***gvm*::*SetRotation*** event with ***view*** an update for the rotation portion of the affine transformation in the ***gvm*::*Node3D*** instance with index ***in.index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setRotationCenter*[*SetRotationCenter3D*:*in*][] – This node schedules a ***gvm*::*SetRotationCenter*** event with ***view*** an update for the rotational center portion of the affine transformation in the ***gvm*::*Node3D*** instance with index ***in.index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setScale*[*SetScale3D*:*in*][] – This node schedules a ***gvm*::*SetScale*** event with ***view*** an update for the scale portion of the affine transformation in the ***gvm*::*Node3D*** instance with index ***in.index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setScaleCenter*[*SetScaleCenter3D*:*in*][] – This node schedules a ***gvm*::*SetScaleCenter*** event with ***view*** an update for the scaling center portion of the affine transformation in the ***gvm*::*Node3D*** instance with index ***in.index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setSphereSize*[*SetSphereSize*:*in*][] – This node schedules a ***gvm*::*SetSphereSize*** event with view an event with time stamp ***getTime*()** to change the parameters of the ***gvm*::*Sphere*** instance with identifier ***in.index*** to the parameters specified in *in*.

node:*setTorusSize*[*SetTorusSize*:*in*][] – This node schedules a ***gvm*::*SetTorusSize*** event with view an event with time stamp ***sodl*::*TimeStamp*::*getTime*()** to change the parameters of the ***gvm*::*Torus*** instance with identifier ***in.index*** to the parameters specified in *in*.

node:*setTranslation*[*SetTranslation3D*:*in*][] – This node schedules a ***gvm*::*SetTranslation*** event with ***view*** an update for the translation portion of the affine transformation in the ***gvm*::*Node3D*** instance with index ***in.index***. The time stamp for this scheduled item is ***sodl*::*TimeStamp*::*getTime*()**.

node:*setVertex*[*SetVertex3D*:*in*][] – This node will schedule a ***gvm*::*SetVertex*** event with ***view*** a change to the ***gvm*::*Vertex3D*** instance with index ***in.index*** to the vertex value ***in.get*()**. The time stamp for the scheduled event will be ***sodl*::*TimeStamp*::*getTime*()**.

B.4. GLUT View Manager (gvm) Classes

The GLUT View Manager uses the classes below to actually display information to a GLUT window. They are owned by a **process:*View*** instance which manages them according to the process hierarchy (which **process:*Node*** instances are owned by each other, etc).

B.4.1. gvm::AddNode

This class is derived from the ***gvm*::*Message*** class and is used to schedule the addition of a subnode to a ***gvm*::*View*** or ***gvm*::*Node*** instance.

Parent Classes: **public *gvm*::*Message***

Private Data Members:

gvm*::*object_index* *gvm*::*AddNode*::*nodeObj – Index of the ***gvm*::*Object*** that is to be added to the node list of the destination.

Public Constructors:

gvm*::*AddNode*::*AddNode*(*gvm*::*View*&** *v*, **double** *t*, ***gvm*::*object_index *o*) – This constructor is used to add the ***gvm*::*Node*** instance with index *o* as a subnode to *v*. This addition will occur at time *t*.

gvm*::*AddNode*::*AddNode*(*gvm*::*View*&** *v*, **double** *t*, ***gvm*::*object_index *d*, ***gvm*::*object_index*** *o*) – This constructor is used to add the ***gvm*::*Node*** instance with index *o* as a subnode to the ***gvm*::*Node*** instance with index *d*. This addition will occur at time *t*.

Public Methods:

virtual void ***gvm::AddNode::send*****(void)** – This method is called when the message is to actually be delivered (when the underlying simulation engine is performing incremental fossil collection for time ***gvm::Message::getTime*****()**. In this case, it actually establishes the parent/subordinate relation specified in the constructor used in creating this instance.

B.4.2. gvm::AddShape

Parent Classes: **public** ***gvm::Message***

Private Data Members:

gvm::object_index gvm::AddShape::shapeObj – This is the index of the ***gvm::Object*** instance (it should actually be a ***gvm::Shape*** instance) that is to be added as a subordinate shape to the destination.

Public Constructors:

gvm::AddShape::AddShape**(*****gvm::View&*** *v*, **double** *t*, ***gvm::object_index*** *d*, ***gvm::object_index*** *o*) – This constructor is used to add the ***gvm::Shape*** instance with index *o* as a subordinate shape to the ***gvm::Node*** instance with index *d*. This addition will occur at time *t*.

Public Methods:

virtual void ***gvm::AddShape::send*****(void)** – This method is called when the message is to actually be delivered (when the underlying simulation engine is performing incremental fossil collection for time ***gvm::Message::getTime*****()**. In this case, it actually establishes the parent/subordinate relation specified in the constructor used in creating this instance.

B.4.3. gvm::AddVertex

Parent Classes: **public** ***gvm::Message***

Private Data Members:

gvm::object_index gvm::AddVertex::vertObj – This is the index of the ***gvm::Vertex*** instance that will be added as a subordinate of the destination.

Public Constructors:

gvm::AddVertex::AddVertex*(*gvm::View*& *v*, double *t*, *gvm::object_index d*, *gvm::object_index o*)** – This constructor is used to add the ***gvm::Vector instance with index *o* as a subordinate vertex to the ***gvm::Polygon2D*** or ***gvm::Polygon3D*** instance with index *d*. This addition will occur at time *t*.

Public Methods:

virtual void *gvm::AddVertex::send*(void) – This method is called when the message is to actually be delivered (when the underlying simulation engine is performing incremental fossil collection for time ***gvm::Message::getTime*()**. In this case, it actually establishes the parent/subordinate relation specified in the constructor used in creating this instance.

B.4.4. gvm::Cone

This provides a means of displaying a cone in a GLUT window. The ***::glutSolidCone***(*base*, *height*, *slices*, *stacks*) and ***::glutWireCone***(*base*, *height*, *slices*, *stacks*) routines are called to render the cone.

Parent Class: **public *gvm::Shape3D***

Derived Classes: None

Protected Data Members:

GLdouble *gvm::Cone::base* – Radius of the cone base.

GLdouble *gvm::Cone::height* – Height of the cone.

GLint *gvm::Cone::slices* – Number of radial slices in the cone slices.

GLint *gvm::Cone::stacks* – Number of lateral stacks for the cone.

Public Constructors:

gvm*::*Cone*::*Cone*(*gvm*::*View3D*& *v*, ulong *i*)** – This constructor calls the parent constructor ***gvm*::*Shape3D*(*v*, GVM_Cone, *i*)** and initializes ***base and ***height*** both to 1.0 and ***slices*** and ***stacks*** both to 10.

Public Methods:

virtual void *gvm*::*Cone*::*display*(void) – This method is used to display a cone using ***::glutSolidCone*(*base*, *height*, *slices*, *stacks*)** if mode is **GL_POLYGON** or is or ***::glutWireCone*(*base*, *height*, *slices*, *stacks*)** otherwise.

virtual bool *gvm*::*Cone*::*isType*(*gvm*::*object_type* *t*) – Returns **true** exactly when *t*=**GVM_Cone** or ***gvm*::*Shape3D*::*isType*(*t*)** returns **true**.

virtual void *gvm*::*Cone*::*set*(GLdouble *b*, GLdouble *h*, GLint *sl*, GLint *st*) – Sets ***base*** to *b*, ***height*** to *h*, ***slices*** to *sl*, and ***stacks*** to *s*.

virtual void *gvm*::*Cone*::*setBase*(GLdouble *b*) – Sets ***base*** to *b*.

virtual void *gvm*::*Cone*::*setHeight*(GLdouble *h*) – Sets ***height*** to *h*.

virtual void *gvm*::*Cone*::*setSlices*(GLint *s*) – Sets ***slices*** to *s*.

virtual void *gvm*::*Cone*::*setStacks*(GLint *s*) – Sets ***stacks*** to *s*.

B.4.5. gvm::CreateObject

This class is used to schedule the creation of a new ***gvm*::*Object*** instance within a ***gvm*::*View*** instance. The creation will occur at time ***gvm*::*Message*::*getTime*()**.

Parent Classes: public *gvm*::*Message*

Private Data Members:

gvm::object_type gvm::CreateObject::objType – Type of object to create

Public Constructors:

gvm::CreateObject::CreateObject(gvm::View& *v*, **double** *t*, ***gvm::object_index*** *d*, ***gvm::object_type*** *o*) – This constructor is used for scheduling the creation of a ***gvm::Object*** instance of type *o* with index *d* at time *t*.

Public Methods:

virtual void ***gvm::CreateObject::send*(void)** – This method actually allocates the ***gvm::Object*** instance and inserts it into the owning view's list of objects.

B.4.6. gvm::Cube

This provides a means of displaying a cube in a GLUT window. The ***::glutSolidCube***(*size*) and ***::glutWireCube***(*size*) routines are called to render the cube.

Parent Class: public *gvm::Shape3D*

Derived Classes: None

Protected Data Members:

GLdouble ***gvm::Cube::size*** – Edge length for the cube.

Public Constructors:

gvm::Cube::Cube(gvm::View3D& *v*, **ulong** *i*) – This constructor calls the parent constructor ***gvm::Shape3D***(*v*, **GVM_Cube,** *i*) and initializes ***size*** to 1.0.

Public Methods:

virtual void ***gvm::Cube::display*(void)** – Display a cube using ***::glutSolidCube***(*size*) if mode is **GL_POLYGON** or is or ***::glutWireCube***(*size*) otherwise.

virtual bool ***gvm::Cube::isType(gvm::object_type*** *t*) – Returns **true** exactly when *t*=**GVM_Cube** or ***gvm::Shape3D::isType***(*t*) returns **true**.

virtual void ***gvm::Cube::set*** (**GLdouble** *s*) – Sets ***size*** to *s*.

B.4.7. gvm::Cylinder

The ***gvm::Cylinder*** class provides a means of displaying a cylinder in a GLUT window. Unlike the other solids displayed here, there is no GLUT routine to display a cylinder, so the author wrote one from scratch. It supports rendering modes GL_POINTS, GL_LINES, GL_LINE_STRIP, and GL_TRIANGLES. If the rendering mode is set to any other mode, it is treated as GL_TRIANGLES.

Parent Class: **public** ***gvm::Shape3D***

Derived Classes: None

Protected Data Members:

GLdouble ***gvm::Cylinder::radius*** – Radius of the cylinder.

GLint ***gvm::Cylinder::nsides*** – Number of sides around a ring.

GLdouble ***gvm::Cylinder::length*** – Length of the cylinder.

GLint ***gvm::Cylinder::rings*** – Number of rings around the cylinder.

std::vector<std::vector<std::vector*<GLdouble> > >** ***gvm::Cylinder::vertices – Hold the vertex values so they do not need to be computed every time. They are changed any time the cylinder parameters are changed.

Public Constructors:

gvm::Cylinder::Cylinder(gvm::View3D& *v*, **ulong** *i*) – This constructor calls the parent class constructor ***gvm::Shape3D***(*v*, **GVM_Cylinder,** *i*) and initializes ***radius***, ***length***, ***nsides*** and ***rings*** to 1.0, 1.0, 10 and 10 respectively.

Public Methods:

virtual void ***gvm::Cylinder::display*****(void)** – Display the cylinder in the currently active GLUT window by rendering the points listed in ***vertices*** in the proper order, given the rendering ***mode***.

virtual bool ***gvm::Cylinder::isType(gvm::object_type*** *t*) – This routine returns to the calling routine **true** if *t*=**GVM_Cylinder** or ***gvm::Shape3D::isType***(*t*) returns **true**.

virtual void ***gvm::Cylinder::set*****(GLdouble** *rad*, **GLdouble** *l*, **GLint** *n*, **GLint** *r*) – Sets ***radius***, ***length***, ***nsides*** and ***rings*** to *rad*, *l*, *n* and *r* respectively.

virtual void ***gvm::Cylinder::setRadius*****(GLdouble** *rad*) – Sets ***radius*** to *rad*.

virtual void ***gvm::Cylinder::setNSides*****(GLint** *n*) – Sets ***nsides*** to *n*.

virtual void ***gvm::Cylinder::setLength*****(GLdouble** *l*) – Sets ***length*** to *l*.

virtual void ***gvm::Cylinder::setRings*****(GLint** *r*) – Sets ***rings*** to *r*.

B.4.8. gvm::Dodecahedron

This provides a means of displaying a dodecahedron in a GLUT window. The ***::glutSolidDodecahedron*****()** and ***::glutWireDodecahedron*****()** routines are called to render the dodecahedron.

Parent Class: **public** ***gvm::Shape3D***

Derived Classes: None

Public Constructors:

gvm::Dodecahedron::Dodecahedron (***gvm::View3D&*** *v*, **ulong** *i*) – This constructor calls the parent constructor ***gvm::Shape3D***(*v*, **GVM_Dodecahedron,** *i*).

Public Methods:

virtual void ***gvm::Dodecahedron::display***(**void**) – Display a dodecahedron using **::*glutSolidDodecahedron*()** if mode is **GL_POLYGON** or **::*glutWireDodecahedron*()** otherwise.

virtual bool ***gvm::Dodecahedron::isType*(*gvm::object_type*** *t*) – Returns **true** exactly when *t*=**GVM_Dodecahedron** or ***gvm::Shape3D::isType***(*t*) returns **true**.

B.4.9. gvm::Icosahedron

This provides a means of displaying an icosahedron in a GLUT window. The **::*glutSolidIcosahedron*()** and **::*glutWireIcosahedron*()** routines are called to render the icosahedron.

Parent Class: **public** ***gvm::Shape3D***

Derived Classes: None

Public Constructors:

gvm::Icosahedron::Icosahedron (***gvm::View3D&*** *v*, **ulong** *i*) – This constructor calls the parent constructor ***gvm::Shape3D***(*v*, **GVM_Icosahedron,** *i*).

Public Methods:

virtual void ***gvm::Icosahedron::display***(**void**) – Display a dodecahedron using **::*glutSolidIcosahedron*()** if mode is **GL_POLYGON** or **::*glutWireIcosahedron*()** otherwise.

virtual bool ***gvm::Icosahedron::isType*(*gvm::object_type*** *t*) – Returns **true** exactly when *t*=**GVM_Icosahedron** or ***gvm::Shape3D::isType***(*t*) returns **true**.

B.4.10. gvm::Message

This is the parent class for all of the messages. These messages are scheduled to occur at some time. They are processed during the fossil collection phase of the simulation and are intended to provide a mechanism to buffer change requests to the scene graph in the graphics system.

Parent Class: **public** ***sodl::Trace***

Derived Classes: ***gvm::AddNode***, ***gvm::AddShape***, ***gvm::AddVertex***, ***gvm::CreateObject***, ***gvm::Refresh***, ***gvm::SetActive***, ***gvm::SetColor***, ***gvm::SetConeSize***, ***gvm::SetCubeSize***, ***gvm::SetCylinderSize***, ***gvm::SetLabel***, ***gvm::SetMode***, ***gvm::SetPointSize***, ***gvm::SetPosition***, ***gvm::SetPosition***, ***gvm::SetRotation***, ***gvm::SetRotationCenter***, ***gvm::SetScale***, ***gvm::SetScaleCenter***, ***gvm::SetSize***, ***gvm::SetSphereSize***, ***gvm::SetTorusSize***, ***gvm::SetTranslation***, ***gvm::SetVertex***

Private Data Members:

gvm::View& gvm::Message::view – This is a reference to owning view.

double ***gvm::Message::time*** – This is the message timestamp

gvm::message_type gvm::Message::type – This is the message type

gvm::object_index gvm::Message::dest – The index of the message destination object.

ulong ***gvm::Message::msgIndex*** – A unique identifier for each message instance associated with a particular ***gvm::View*** instance.

Public Constructors:

gvm::Message::Message(gvm::View& v, **double** ***t, gvm::message_type ty, gvm::object_index i)*** – This constructor initializes ***view*** to *v*, ***time*** to *t*, ***type*** to *ty* and ***dest*** to *i*.

Public Methods:

virtual ***gvm::object_index gvm::Message::getDest*****(void) const** – This method returns ***dest*** to the calling routine.

virtual double ***gvm::Message::getTime*****(void) const** – This method returns ***time*** to the calling routine

virtual ***gvm::message_type gvm::Message::getType*****(void) const** – This method returns ***type*** to the calling routine.

virtual ***gvm*::View& *gvm*::*Message*::*getView*(void)** – This method returns ***view*** to the calling routine.

virtual void *gvm*::*Message*::*send*(void) – Derived classes overload this method to perform the specific functions associated with delivering the message.

virtual void *gvm*::*Message*::*setIndex*(ulong *i*) – This method sets ***msgIndex*** to *i*.

virtual ulong *gvm*::*Message*::*getIndex*(void) – Return ***msgIndex*** to the calling routine.

B.4.11. gvm::Node

Parent Class: public *gvm*::*Object*

Derived Classes: ***gvm*::*Node2D*, *gvm*::*Node3D***

Protected Enumerators:

enum *gvm*::*Node*::*node_flags*{NF_Color, NF_PointSize, NF_LAST} – These flags are used to index an array of bool values associated with various flag values.

Protected Data Members:

std*::*vector*<GLdouble> *gvm*::*Message*::*color – When ***flags*[NF_Color]** is set to **true**, then the current drawing color is saved and the color specified in this data member is used instead. When the subordinate objects are finished being rendered, the original color is restored for additional processing.

std*::*vector*<GLdouble> *gvm*::*Node*::*ctrRot – Specifies the center of rotation for this rendering node.

std*::*vector*<GLdouble> *gvm*::*Node*::*ctrScale – Specifies the center of scaling for this rendering node.

std*::*vector*<bool> *gvm*::*Message*::*flags – This array contains flags for either using the local values for color and point size or to use the default values in place when the ***display*** method is called. When the flag is set to **true**, then the local value is used for all subordinates. Otherwise, the current value in effect at the calling of the ***display*** method is used.

GLfloat ***gvm*****::*****Message*****::*****pt_size*** – When ***flags***[**NF_PointSize**] is set to **true**, then the current point size parameter is saved and the size specified in this data member is used instead. When the subordinate objects are finished being rendered, the original point size parameter is restored for additional processing.

bool ***gvm*****::*****Node*****::*****running*** – For detecting preventing infinite loops. This gets set to **true** when rendering for this node starts, and false when it's done. If it is asked to render itself while **true**, the request is ignored without performing any rendering.

std**::*****vector*****<GLdouble>** ***gvm*****::*****Node*****::*****rot*** – Specifies the rotation angle for this rendering node.

std**::*****vector*****<GLdouble>** ***gvm*****::*****Node*****::*****scale*** – Specifies the scaling factors for this rendering node.

std**::*****vector<gvm*****::*****Object*>*** ***gvm*****::*****Node*****::*****subordinateList*** – This acts as a list of subordinate components. It mixes both subordinate nodes and shapes in the same list.

std**::*****vector*****<GLdouble>** ***gvm*****::*****Node*****::*****trans*** – Specifies the translation factors for this rendering node.

Protected Constructors:

gvm**::*****Node*****::*****Node(gvm*****::*****View&*** *v*, ***gvm*****::*****object_type*** *t*, ***gvm*****::*****object_index*** *i*) – This constructor calls the parent constructor ***gvm*****::*****Object***(*v*, *t*, *i*) and initializes the other data members. Each of the affine transformation components are initialized to arrays of size two or three, for *t* **GVM_Node2D** and **GVM_Node3D** respectively, at the origin (except ***scale***, which is at <1, 1> or <1,1,1> as appropriate). Members ***running*** and ***pt_size*** are initialized to **false** and 1.0 respectively. Arrays ***color*** and ***flags*** are set to <-1, -1, -1, -1> and <**false**, **false**, **false**> respectively.

Public Constructors:

Public Methods:

virtual void ***gvm*****::*****Node*****::*****addObject(gvm*****::*****object_index*** *i*) – This routine adds the pointer associated with the reference ***gvm*****::*****Object*****::*****getView***()[*i*] to ***subordinateList***.

virtual void *gvm*::*Node*::***display***(**void**) – If ***flags***[**NF_Color**] is **true** then the current drawing color is saved and reset to **color**. Likewise, if ***flags***[**NF_PointSize**] is **true**, the current point size parameter is saved and the default point size is set to ***pt_size***. It then renders the scene from this node to all of its subordinate objects. The previous drawing color and point size are then restored prior to returning to the calling routine.

virtual bool *gvm*::*Node*::***isType***(*gvm*::***object_type*** *t*) – This method returns **true** when either *t*=**GVM_Node** or *gvm*::***Object***::***isType***(*t*) returns **true**.

virtual void *gvm*::*Node*::***setColor***(**GLdouble** *r*, **GLdouble** *g*, **GLdouble** *b*) – Sets data member color to ::***make_vector***(4, *r*, *g*, *b*, 1.0). If *r*, *g*, and *b* are all in the range [0.0, 1.0] then ***flags***[**NF_Color**] is set to **true**; **false** otherwise.

virtual void *gvm*::*Node*::***setColor***(**GLdouble** *r*, **GLdouble** *g*, **GLdouble** *b*, **GLdouble** *a*) – Sets data member color to ::***make_vector***(4, *r*, *g*, *b*, *a*). If *r*, *g*, *b* and *a* are all in the range [0.0, 1.0] then ***flags***[**NF_Color**] is set to **true**; **false** otherwise.

virtual void *gvm*::*Node*::***setColor***(*std*::*vector*<**GLdouble**> *c*) – This method sets ***color*** to ::***resize_vector***(4, 1.0, *c*). If all of the elements of *c* are in the range [0.0, 1.0], then ***flags***[**NF_Color**] is set to **true**; **false** otherwise.

virtual void *gvm*::*Node*::***setPointSize***(**GLfloat** *s*) – This method sets ***pt_size*** to *s*. If *s*≥0.0 then ***flags***[**NF_PointSize**] is set to **true**; **false** otherwise.

virtual void *gvm*::*Node*::***setRotation***(**const** *std*::*vector*<**GLdouble**> **&** *v*) – Sets data member ***rot*** to ::***resize_vector***(***rot.size***(), 0.0, *v*).

virtual void *gvm*::*Node*::***setRotationCenter***(**const** *std*::*vector*<**GLdouble**> **&** *v*) – Sets data member ***ctrRot*** to ::***resize_vector***(***ctrRot.size***(), 0.0, *v*).

virtual void *gvm*::*Node*::***setScaleCenter***(**const** *std*::*vector*<**GLdouble**> **&** *v*) – Sets data member ***ctrScale*** to ::***resize_vector***(***ctrScale.size***(), 0.0, *v*).

virtual void ***gvm*::*Node*::*setScale*(const** ***std*::*vector*<GLdouble> &** *v*) – Sets data member ***scale*** to ::***resize_vector*(*scale*.*size*()**, 1.0, *v*).

virtual void ***gvm*::*Node*::*setTranslation*(const** ***std*::*vector*<GLdouble> &** *v*) – Sets date member ***trans*** to ::***resize_vector*(*trans*.*size*()**, 1.0, *v*).

B.4.12. gvm::Node2D

This specializes the node to perform two-dimensional affine transformations.

Parent Class: **public** ***gvm*::*Node***

Derived Classes: None

Public Constructors:

gvm*::*Node2D*::*Node2D*(*gvm*::*View2D& *v*, **ulong** *i*) – This constructor calls the parent constructor ***gvm*::*Node*(*v*, GVM_Node2D,** *i*).

Public Methods:

virtual void ***gvm*::*Node2D*::*addNode*(*gvm*::*object_index*** *n*) – Add an existing node, given by ***gvm*::*Object*::*getView*()[*n*]** instance to the list of subordinate objects.

virtual void ***gvm*::*Node2D*::*addShape*(*gvm*::*object_index*** *s*) – Add an existing shape, given by ***gvm*::*Object*::*getView*()[*n*]** instance to the list of subordinate objects.

virtual void ***gvm*::*Node2D*::*display*(void)** – Display routine for this node. It performs the node's affine transformations and then calls the parent version ***gvm*::*Node*::*display*()** to actually display the subordinate objects.

virtual bool ***gvm*::*Node2D*::*isType*(*gvm*::*object_type*** *t*) – Returns **true** exactly when *t*=**GVM_Node2D** or ***gvm*::*Node*::*isType*(*t*)** returns **true**.

virtual void ***gvm*::*Node2D*::*setRotation***(**GLdouble** *z*) – Sets the data member ***gvm*::*Node*::*rot*** to **::*make_vector***(2, *z*, 0.0).

virtual void ***gvm*::*Node2D*::*setRotationCenter***(**GLdouble** *x*, **GLdouble** *y*) – Sets the data member ***gvm*::*Node*::*ctrRot*** to **::*make_vector***(2, *x*, *y*).

virtual void ***gvm*::*Node2D*::*setScale***(**GLdouble** *x*, **GLdouble** *y*) – Sets the data member ***gvm*::*Node*::*scale*** to **::*make_vector***(2, *x*, *y*).

virtual void ***gvm*::*Node2D*::*setScaleCenter***(**GLdouble** *x*, **GLdouble** *y*) – Sets the data member ***gvm*::*Node*::*ctrScale*** to **::*make_vector***(2, *x*, *y*).

virtual void ***gvm*::*Node2D*::*setTranslation***(**GLdouble** *x*, **GLdouble** *y*) – Sets the data member ***gvm*::*Node*::*trans*** to **::*make_vector***(2, *x*, *y*).

B.4.13. gvm::Node3D

This specializes the node to perform three-dimensional affine transformations.

Parent Class: **public *gvm*::*Node***

Derived Classes: None

Public Constructors:

gvm*::*Node3D*::*Node3D(***gvm*::*View3D&*** *v*, **ulong** *i*) – This constructor calls the parent constructor ***gvm*::*Node***(*v*, **GVM_Node3D**, *i*).

Public Methods:

virtual void ***gvm*::*Node3D*::*addNode***(***gvm*::*object_index*** *n*) – Add an existing node, given by ***gvm*::*Object*::*getView***()[*n*] instance to the list of subordinate objects.

virtual void ***gvm*::*Node3D*::*addShape***(***gvm*::*object_index*** *s*) – Add an existing shape, given by ***gvm*::*Object*::*getView***()[*n*] instance to the list of subordinate objects.

virtual void ***gvm*::*Node3D*::*display*(void)** – Display routine for this node. It performs the node's affine transformations and then calls the parent version ***gvm*::*Node*::*display*()** to actually display the subordinate objects.

virtual bool *gvm*::*Node3D*::*isType*(*gvm*::*object_type t*) – Returns **true** exactly when *t*=**GVM_Node3D** or *gvm*::*Node*::*isType*(*t*) returns **true**.

virtual void *gvm*::*Node3D*::*setRotation*(**GLdouble** *z*) – Sets the data member ***gvm*::*Node*::*rot*** to ::***make_vector***(2, *z*, 0.0).

virtual void *gvm*::*Node3D*::*setRotationCenter*(**GLdouble** *x*, **GLdouble** *y*, **GLdouble** *z*) – Sets the data member ***gvm*::*Node*::*ctrRot*** to **::*make_vector***(3, *x*, *y*, *z*).

virtual void *gvm*::*Node3D*::*setScale*(**GLdouble** *x*, **GLdouble** *y*, **GLdouble** *z*) – Sets the data member ***gvm*::*Node*::*scale*** to **::*make_vector***(3, *x*, *y*, *z*).

virtual void *gvm*::*Node3D*::*setScaleCenter*(**GLdouble** *x*, **GLdouble** *y*, **GLdouble** *z*) – Sets the data member ***gvm*::*Node*::*ctrScale*** to **::*make_vector***(3, *x*, *y*, *z*).

virtual void *gvm*::*Node3D*::*setTranslation*(**GLdouble** *x*, **GLdouble** *y*, **GLdouble** *z*) – Sets the data member ***gvm*::*Node*::*trans*** to **::*make_vector***(3, *x*, *y*, *z*).

B.4.14. gvm::Object

This is the base class for all of the ***gvm*::** classes displayed in ***gvm*::*View*** instances. It provides some basic mechanisms for displaying information to the ***gvm*::*View*** instances. It contains many of the methods for setting data members within derived classes. This allows a certain level of abstraction that is useful for delivering buffered messages to ***gvm*::*Object*** instances that are not of any predefined type. If a message does something to a ***gvm*::*Object*** instance that does not make any sense, the instance will allow the call, but it will be ignored, and a warning message will be delivered to ***std*::*out***.

Parent Class: public *sodl*::*Trace*

Derived Classes: ***gvm*::*Node***, ***gvm*::*Shape***, ***gvm*::*Vertex***

Private Data Members:

gvm*::*View** *gvm*::*Object*::*view – Pointer to the ***gvm*::*View*** instance to which this ***gvm*::*Object*** instance is subordinate.

gvm*::*object_handle* *gvm*::*Object*::*handle – This is a unique identifier associated with this specific ***gvm*::*Object*** instance.

std*::*string* *gvm*::*Object*::*label – A settable label for identification purposes.

Protected Data Members:

bool ***gvm*::*Object*::*active*** – This is set to **true** exactly when this ***gvm*::*Object*** instance is active. When set to **true**, it will enable the object to be displayed; when **false**, the code in the ***display*** method is to be ignored.

Public Constructors:

gvm*::*Object*::*Object*(*gvm*::*View*& *v*, *gvm*::*object_type* *t*, *gvm*::*object_index* *i*)** – This constructor initializes ***view to *v*, ***handle*** to (*t*, *i*), and ***active*** and ***label*** to **true** and "none" respectively.

Public Methods:

virtual void ***gvm*::*Object*::*addNode*(*gvm*::*object_index*)** – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*addShape*(*gvm*::*object_index*)** – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*addVertex*(*gvm*::*object_index*)** – A placeholder for some derived classes to overload.

virtual void ***gvm::Object::begin*****(void)** – A placeholder for some derived classes to overload.

virtual void ***gvm::Object::display*****(void)** – This method should be overloaded to perform output specific to the derived class instance.

virtual void ***gvm::Object::end*****(void)** – A placeholder for some derived classes to overload.

virtual ***gvm::object_handle gvm::Object::getHandle*****(void)** – Return ***handle*** to the calling routine.

virtual ***gvm::object_index gvm::Object::getIndex*****(void)** – Return ***handle.second*** to the calling routine.

virtual ***std::string gvm::Object::getLabel*****(void)** – Return ***label*** to the calling routine.

virtual ***gvm::object_type gvm::Object::getType*****(void)** – Return ***handle.first*** to the calling routine.

virtual ***gvm::View& gvm::Object::getView*****(void)** – Returns ***view*** to the calling routine.

virtual bool ***gvm::Object::isActive*****(void) const** – Return ***active*** to the calling routine.

virtual bool ***gvm::Object::isType(gvm::object_type t*****)** – Return **true** exactly when *t*=**GVM_Object**.

virtual void ***gvm::Object::set*****(GLdouble)** – A placeholder for some derived classes to overload.

virtual void ***gvm::Object::set*****(GLdouble, GLdouble)** – A placeholder for some derived classes to overload.

virtual void ***gvm::Object::set*****(GLdouble, GLdouble, GLdouble)** – A placeholder for some derived classes to overload.

virtual void ***gvm::Object::set*****(GLdouble, GLdouble, GLint, GLint)** – A placeholder for some derived classes to overload.

virtual void ***gvm::Object::set*****(GLdouble, GLint, GLint)** – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*set***(**const** ***std*::*vector*<GLdouble>&**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setActive***(**bool** *a*) – Sets the ***active*** flag to *a*.

virtual void ***gvm*::*Object*::*setBase***(**GLdouble**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setColor***(**GLdouble, GLdouble, GLdouble**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setColor***(**GLdouble, GLdouble, GLdouble, GLdouble**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setColor*(*std*::*vector*<GLdouble>**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setHeight***(**GLdouble**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setInnerRadius***(**GLdouble**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setLabel*(*std*::*string*** *s*) – Set ***label*** to *s*.

virtual void ***gvm*::*Object*::*setMode***(**GLenum m**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setNSides***(**GLint**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setOuterRadius***(**GLdouble**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setPointSize***(**GLfloat**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setRadius***(**GLdouble**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setRings***(**GLint**) – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setRotation*(GLdouble)** – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setRotation*(GLdouble, GLdouble, GLdouble)** – A place holder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setRotation*(const *std*::*vector*<GLdouble>&)** – A place holder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setRotationCenter*(GLdouble, GLdouble)** – A placeholder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setRotationCenter*(GLdouble, GLdouble, GLdouble)** – A place holder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setRotationCenter*(const *std*::*vector*<GLdouble>&)** – A place holder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setScale*(GLdouble, GLdouble)** – A place holder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setScale*(GLdouble, GLdouble, GLdouble)** – A place holder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setScale*(const *std*::*vector*<GLdouble>&)** – A place holder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setScaleCenter*(GLdouble, GLdouble)** – A place holder for some derived classes to overload.

virtual void ***gvm*::*Object*::*setScaleCenter*(GLdouble, GLdouble, GLdouble)** – A place holder for some derived classes to overload.

virtual void ***gvm*****::*****Object*****::*****setScaleCenter*****(const** ***std*****::*****vector*****<GLdouble>&)** – A place holder for some derived classes to overload.

virtual void ***gvm*****::*****Object*****::*****setSlices*****(*****GLint*****)** – A place holder for some derived classes to overload.

virtual void ***gvm*****::*****Object*****::*****setStacks*****(GLint)** – A place holder for some derived classes to overload.

virtual void ***gvm*****::*****Object*****::*****setTranslation*****(GLdouble, GLdouble)** – A place holder for some derived classes to overload.

virtual void ***gvm*****::*****Object*****::*****setTranslation*****(GLdouble, GLdouble, GLdouble)** – A place holder for some derived classes to overload.

virtual void ***gvm*****::*****Object*****::*****setTranslation*****(const** ***std*****::*****vector*****<GLdouble>&)** – A place holder for some derived classes to overload.

virtual void ***gvm*****::*****Object*****::*****setView*****(*****gvm*****::*****View*****&** *v*) – Sets ***view*** to *v*.

B.4.15. gvm::Octahedron

This provides a means of displaying an octahedron in a GLUT window. The **::*****glutSolidOctahedron*****()** and **::*****glutWireOctahedron*****()** routines are called to render the octahedron.

Parent Class: **public** ***gvm*****::*****Shape3D***

Derived Classes: None

Public Constructors:

gvm**::*****Octahedron*****::*****Octahedron*****(*****gvm*****::*****View3D*****&** *v*, **ulong** *i*) – This constructor calls the parent constructor ***gvm*****::*****Shape3D*****(*****v*****, GVM_Octahedron,** ***i*****)**.

Public Methods:

virtual void ***gvm::Octahedron::display*(void)** – This method displays an octahedron using ***::glutSolidOctahedron*()** if ***mode***=**GL_POLYGON** or ***::glutWireOctahedron*()** otherwise.

virtual bool ***gvm::Octahedron::isType(gvm::object_type t)*** – This routine returns to the calling routine **true** if *t*=**GVM_Octahedron** or ***gvm::Shape3D::isType*(*t*)** returns **true**.

B.4.16. gvm::Polygon2D

This class is used for displaying groups of two-dimensional vertices to the parent ***gvm::View2D*** window. When mode is **GL_POLYGON**, the vertices need to form a convex polygon.

Parent Class: **public *gvm::Shape2D***

Derived Classes: None

Protected Methods:

std::vector<gvm::Object*> gvm::Polygon2D::vertList – List of vertices to display to the parent view.

Public Constructors:

gvm::Polygon2D::Polygon2D(gvm::View2D& v,* ulong *i) – This constructor calls the parent constructor ***gvm::Shape2D*(*v*, GVM_Polygon2D, *i*)**.

Public Methods:

virtual void ***gvm::Polygon2D::display*(void)** – Display this two-dimensional polygon to the currently active GLUT window.

virtual void ***gvm::Polygon2D::addVertex(gvm::object_index i)*** – This method will add ***&gvm::Object::getView*()[*i*]** to the back of ***vertList***.

B.4.17. gvm::Polygon3D

This class is used for displaying groups of three-dimensional vertices to the parent ***gvm::View3D*** window. When mode is **GL_POLYGON**, the vertices need to form a convex polygon.

Parent Class: **public** ***gvm::Shape3D***

Derived Classes: None

Protected Methods:

std::vector<gvm::Object*> gvm::Polygon3D::vertList – List of vertices to display to the parent view.

Public Constructors:

gvm::Polygon3D::Polygon3D(gvm::View3D& *v*, **ulong** *i*) – This constructor calls the parent constructor ***gvm::Shape3D***(*v*, **GVM_Polygon3D**, *i*).

Public Methods:

virtual void ***gvm::Polygon3D::display***(**void**) – Display this two-dimensional polygon to the currently active GLUT window.

virtual void ***gvm::Polygon3D::addVertex(gvm::object_index*** *i*) – This method will add ***&gvm::Object::getView***()[*i*] to the back of ***vertList***.

B.4.18. gvm::Refresh

The ***gvm::Refresh*** message is used to schedule a screen refresh. Once it has been scheduled, the refresh is actually performed during fossil collection of the owning **process:*View*** instance.

Parent Classes: **public** ***gvm::Message***

Derived Classes: None

Public Constructors:

gvm::Refresh::Refresh(gvm::View&* *v*, **double** *t*) – This class constructor calls the parent constructor ***gvm::Message(*v*, *t*, **GVM_Refresh, (ulong) -1**).

Public Methods:

virtual void *gvm::Message::send*(void) – This method sets the refresh flag in ***view*** so that the next fossil collection event will force a screen refresh.

B.4.19. gvm::SetActive

The ***gvm::SetActive*** message sets the active flag for the destination ***gvm::Object*** instance, turning it either on or off within ***view***.

Parent Classes: public *gvm::Message*

Derived classes: None

Private Data Members:

bool *gvm::SetActive::active* – This contains the value for the active flag in the destination object.

Public Constructors:

gvm::SetActive::SetActive(gvm::View&* *v*, double *t*, *gvm::object_index* *d*, bool *a*)** – Class constructor which calls parent constructor ***gvm::Message(*v*, *t*, **GVM_SetActive,** *i*) and initializes ***active*** to *a*.

Public Methods:

virtual void *gvm::SetActive::send*(void) – Sets the active flag of ***getView***()[***dest***] to ***active***.

B.4.20. gvm::SetColor

The ***gvm::SetColor*** message is intended to change the color attribute of the destination ***gvm::Object*** instance, either a ***gvm::Shape*** or ***gvm::Node*** instance.

Parent Classes: public *gvm::Message*

Derived Classes: None

Private Data Members:

std*::*vector*<GLdouble> *gvm*::*SetColor*::*color – Color to set the destination object.

Public Constructors:

gvm*::*SetColor*::*SetColor*(*gvm*::*View*&** *v*, **double** *t*, ***gvm*::*object_index *i*, ***std*::*vector*<GLdouble>** *c*) – This constructor initializes ***color*** to ***c*** and calls the parent constructor ***gvm*::*Message*(***v*, *t*, **GVM_SetColor,** *i***)**.

Public Methods:

virtual void *gvm*::*SetColor*::*send*(void) – This method sets the color attribute of the destination object to ***color***.

B.4.21. gvm::SetConeSize

The ***gvm*::*SetConeSize*** message is intended to change the cone size attributes of the destination ***gvm*::*Cone*** instance.

Parent Classes: public *gvm*::*Message*

Derived Classes: None

Private Data Members:

GLdouble *gvm*::*SetConeSize*::*base* – Size to set the base attribute of the destination ***gvm*::*Cone*** instance.

GLdouble *gvm*::*SetConeSize*::*height* – Size to set the height attribute of the destination ***gvm*::*Cone*** instance.

GLint *gvm*::*SetConeSize*::*slices* – Number of radial slices composing the destination ***gvm*::*Cone*** instance.

GLint ***gvm::SetConeSize::stacks*** – Number of lateral slices composing the destination ***gvm::Cone*** instance.

Public Constructors:

gvm::SetConeSize::SetConeSize(gvm::View& *v*, **double** *t*, ***gvm::object_index*** *i*, **GLdouble** *b*, **GLdouble** *h*, **GLint** *sl*, **GLint** *st*) – This constructor calls the parent constructor ***gvm::Message***(*v*, *t*, **GVM_SetConeSize,** *i*) and initializes ***base*** to *b*, ***height*** to *h*, ***slices*** to *sl* and ***stacks*** to *st*.

Public Methods:

virtual void ***gvm::SetConeSize::send***(**void**) – This method will update the parameters of the destination ***gvm::Cone*** instance to the parameters in the payload of this message.

B.4.22. gvm::SetCubeSize

The ***gvm::SetCubeSize*** message is sent to ***gvm::Cube*** instances to change the size of the cube edges.

Parent Classes: public ***sodl::Message***

Derived Classes: None

Private Data Members:

GLdouble ***gvm::SetCubeSize::cube_size*** – New size attribute for the destination ***gvm::Cube*** instance.

Public Constructors:

gvm::SetCubeSize::SetCubeSize(gvm::View& *v*, **double** *t*, ***gvm::object_index*** *i*, **GLdouble** *s*) – This constructor initializes ***cube_size*** to *s* and calls the parent constructor ***gvm::Message***(*v*, *t*, **GVM_SetCubeSize,** *i*).

Public Methods:

virtual void ***gvm::SetCubeSize::send*****(void)** – This methods actually sets the size attribute of the destination ***gvm::Cube*** instance.

B.4.23. gvm::SetCylinderSize

The ***gvm::SetCylinderSize*** message is intended to set the size attributes of a ***gvm::Cylinder*** instance.

Parent Classes: public ***gvm::Message***

Derived Classes: None

GLdouble ***gvm::SetCylinderSize::radius*** – Value to set the radius attribute of the destination ***gvm::Cylinder*** instance.

GLdouble ***gvm::SetCylinderSize::length*** – Value to set the length attribute of the destination ***gvm::Cylinder*** instance.

GLint ***gvm::SetCylinderSize::sides*** – Value to set the side count attribute of the destination ***gvm::Cylinder*** instance.

GLint ***gvm::SetCylinderSize::rings*** – Value to set the ring count attribute of the destination ***gvm::Cylinder*** instance.

Public Constructors:

gvm::SetCylinderSize::SetCylinderSize**(*****gvm::View&*** ***v*****, double** ***t*****,** ***gvm::object_index i*****, GLdouble** ***ir*****, GLdouble** ***or*****, GLint** ***s*****, GLint** ***r*****)** – This constructor calls the parent class constructor ***gvm::Message*****(*****v*****,** ***t*****, GVM_SetCylinderSize,** ***i*****)** and initializes ***innerRadius***, ***outerRadius***, ***sides***, and ***rings*** to *ir*, *or*, *s*, and *r* respectively.

Public Method:

virtual void ***gvm::SetCylinderSize::send*****(void)** – This method commits the changes in the various attributes of the destination ***gvm::Cylinder*** instance.

B.4.24. gvm::SetLabel

The ***gvm::SetLabel*** message is used to set the label attribute of the destination ***gvm::Object***.

Parent Classes: public *gvm::Message*

Derived Classes: None

Private Data Members:

std::string gvm::SetLabel::label – Value to set the label attribute of the destination ***gvm::Object*** instance.

Public Constructors:

gvm::SetLabel::SetLabel(gvm::View& *v*, **double** *t*, ***gvm::object_index*** *i*, ***std::string*** *l*) – This constructor calls the parent constructor ***gvm::Message***(*c*, *t*, **GVM_SetLabel,** *i*) and initializes ***label*** to *l*.

Public Methods:

virtual void *gvm::SetLabel::send*(void) – This method actually sets the label attribute of the destination object.

B.4.25. gvm::SetMode

The ***gvm::SetMode*** message is used to set the mode attribute of some ***gvm::Object*** instances.

Parent Classes: public *gvm::Message*

Derived Classes: None

Private Data Members:

GLenum *gvm::SetMode::mode* – Value to set the mode attribute of the destination object.

Public Constructors:

gvm*::*SetMode*::*SetMode*(*gvm*::*View*& *v*, double *t*, *gvm*::*object_index* *i*, GLenum *m*)** – This constructor calls the parent constructor ***gvm*::*Message*(*v*, *t*, GVM_SetMode, *i*)** and initialize ***mode to ***m***.

Public Methods:

virtual void *gvm*::*SetMode*::*send*(void) – This method sets the mode parameter of the destination ***gvm*::*Object*** instance.

B.4.26. gvm::SetPointSize

The ***gvm*::*SetPointSize*** message is used to set the point size attribute within destination ***gvm*::*Object*** instances.

Parent Classes: **public *gvm*::*Message***

Derived Classes: None

Private Data Members:

double *gvm*::*SetPointSize*::*point_size* – Value to set the point size attribute in the destination ***gvm*::*Object*** instance or ***gvm*::*View*** instance, as appropriate.

Public Constructors:

gvm*::*SetPointSize*::*SetPointSize*(*gvm*::*View*& *v*, double *t*, double *ps*)** – This constructor calls the parent constructor ***gvm*::*Message*(*v*, *t*, GVM_SetPointSize, (ulong) –1)** and initializes ***point_size to *ps*. This constructor is used when the intended destination of the message is ***view***.

gvm*::*SetPointSize*::*SetPointSize*(*gvm*::*View*& *v*, double *t*, *gvm*::*object_index* *i*, double *ps*)** – This constructor initializes ***point_size to *ps* and calls the parent constructor ***gvm*::*Message*(*v*, *t*, GVM_SetPointSize, *i*)**. This constructor is used when the intended destination is the ***gvm*::*Object*** instance with identifier *i*.

Public Methods:

virtual void ***gvm*::*SetPointSize*::*send*(void)** – This method actually commits the change to the point size attribute of the destination ***gvm*::*Object*** or ***gvm*::*View*** instance.

B.4.27. gvm::SetPosition

The ***gvm*::*SetPosition*** message is intended to set the position of the GLUT window.

Parent Classes: **public** ***gvm*::*Message***

Derived Classes: None

Private Data Members:

GLint ***gvm*::*SetPosition*::*x*** – New X location of the GLUT window.

GLint ***gvm*::*SetPosition*::*y*** – New Y location of the GLUT window.

Public Class Constructors:

gvm*::*SetPosition*::*SetPosition*(*gvm*::*View*& *v*, double *t*, GLint *X*, GLint *Y*)** – This constructor initializes ***x and ***y*** to *X* and *Y* respectively and calls the parent constructor ***gvm*::*Message*(*v*, *t*, *GVM_SetPosition*, (ulong) –1)**.

Public Methods:

virtual void ***gvm*::*SetPosition*::*send*(void)** – This method will set the window position of the destination ***gvm*::*View*** instance to <*x*, *y*>.

B.4.28. gvm::SetRotation

The ***gvm*::*SetRotation*** message is intended to set the rotation attribute of a ***gvm*::*Node*** instance.

Parent Classes: **public** ***gvm*::*Message***

Derived Classes: None

Private Data Members:

std*::*vector*<GLdouble> *gvm*::*SetRotation*::*rot – Value to set the rotation attribute of the destination ***gvm*::*Node*** instance.

Public Constructors:

gvm*::*SetRotation*::*SetRotation*(*gvm*::*View*& *v*, double *t*, *gvm*::*object_index* *i*, *std*::*vector*<GLdouble> *r*)** – This constructor initializes ***rot to *r* and calls the parent class constructor ***gvm*::*Message*(*v*, *t*, GVM_SetRotation, *i*)**.

Public Methods:

virtual void *gvm*::*SetRotation*::*send*(void) – This method actually sets the rotation attribute of the destination ***gvm*::*Node*** instance.

B.4.29. gvm::SetRotationCenter

The ***gvm*::*SetRotationCenter*** message is intended to set the center of rotation attribute of a ***gvm*::*Node*** instance.

Parent Classes: **public *gvm*::*Message***

Derived Classes: None

Private Data Members:

std*::*vector*<GLdouble> *gvm*::*SetRotationCenter*::*ctrRot – Value to set the rotation center attribute of the destination ***gvm*::*Node*** instance.

Public Constructors:

gvm*::*SetRotationCenter*::*SetRotationCenter*(*gvm*::*View*& *v*, double *t*, *gvm*::*object_index* *i*, *std*::*vector*<GLdouble> *c*)** – This constructor initializes ***ctrRot to *c* and calls the parent class constructor ***gvm*::*Message*(*v*, *t*, GVM_SetRotationCenter, *i*)**.

Public Methods:

virtual void *gvm*::*SetRotationCenter*::*send*(void) – This method actually sets the center of rotation attribute of the destination ***gvm*::*Node*** instance.

B.4.30. gvm::SetScale

The ***gvm*::*SetScale*** message is intended to set the scale attribute of a ***gvm*::*Node*** instance.

Parent Classes: **public *gvm*::*Message***

Derived Classes: None

Private Data Members:

std*::*vector*<GLdouble> *gvm*::*SetScale*::*scale – Value to set the scale attribute of the destination ***gvm*::*Node*** instance.

Public Constructors:

gvm*::*SetScale*::*SetScale*(*gvm*::*View*& *v*, double *t*, *gvm*::*object_index i*, *std*::*vector*<GLdouble> *s*)** – This constructor initializes ***scale to *s* and calls the parent class constructor ***gvm*::*Message*(*v*, *t*, GVM_SetScale, *i*)**.

Public Methods:

virtual void *gvm*::*SetScale*::*send*(void) – This method actually sets the scale attribute of the destination ***gvm*::*Node*** instance.

B.4.31. gvm::SetScaleCenter

The ***gvm*::*SetScaleCenter*** message is intended to set the center of scaling attribute of a ***gvm*::*Node*** instance.

Parent Classes: **public *gvm*::*Message***

Derived Classes: None

Private Data Members:

std*::*vector*<GLdouble> *gvm*::*SetScaleCenter*::*ctrScale – Value to set the scaling center attribute of the destination ***gvm*::*Node*** instance.

Public Constructors:

gvm*::*SetScaleCenter*::*SetScaleCenter*(*gvm*::*View*&** *v*, **double** *t*, ***gvm*::*object_index *i*, ***std*::*vector*<GLdouble>** *c*) – This constructor initializes ***ctrScale*** to *c* and calls the parent class constructor ***gvm*::*Message***(*v*, *t*, **GVM_SetScaleCenter,** *i*).

Public Methods:

virtual void *gvm*::*SetScaleCenter*::*send*(void) – This method actually sets the center of scaling attribute of the destination ***gvm*::*Node*** instance.

B.4.32. gvm::SetSize

The ***gvm*::*SetSize*** message is intended to set the GLUT viewport size of the destination ***gvm*::*View*** instance.

Parent Classes: public *gvm*::*Message*

Private Data Members:

GLint gvm*::*SetSize*::*width – Value to set the width attribute of the destination ***gvm*::*View*** instance.

GLint *gvm*::*SetSize*::*height* – Value to set the height attribute of the destination ***gvm*::*View*** instance.

Public Constructors:

gvm*::*SetSize*::*SetSize*(*gvm*::*View*& *v*, double *t*, GLint *w*, GLint *h*)** – This constructor calls the parent constructor ***gvm*::*Message(*v*, *t*, **GVM_SetSize, (ulong) –1)** and initializes ***width*** and ***height*** to *w* and *h* respectively.

Public Methods:

virtual void *gvm*::*SetSize*::*send*(void) – This method commits the GLUT viewport size changes for the intended ***gvm*::*View*** instance.

B.4.33. gvm::SetSphereSize

The ***gvm*::*SetSphereSize*** message is intended to set the size attributes of a ***gvm*::*Sphere*** instance.

Parent Classes: **public *gvm*::*Message***

Derived Classes: None

GLdouble *gvm*::*SetSphereSize*::*radius* – Value to set the radius attribute of the destination ***gvm*::*Sphere*** instance.

GLint *gvm*::*SetSphereSize*::*slices* – Value to set the ***slices*** attribute of the destination ***gvm*::*Sphere*** instance.

GLint *gvm*::*SetSphereSize*::*stacks* – Value to set the ***stacks*** attribute of the destination ***gvm*::*Sphere*** instance.

Public Constructors:

gvm*::*SetSphereSize*::*SetSphereSize*(*gvm*::*View*& *v*, double t, *gvm*::*object_index* *i*, GLdouble *r*, GLint *sl*, GLint *st*)** – This constructor calls the parent constructor ***gvm*::*Message(*v*, *t*, **GVM_SetSphereSize,** *i*) and initializes ***radius***, ***slices*** and ***stacks*** to *r*, *sl*, and *st* respectively.

Public Methods:

virtual void ***gvm::SetSphereSize::send*****(void)** – This method commits the changes to the attributes of the destination ***gvm::Sphere*** instance.

B.4.34. gvm::SetTorusSize

The ***gvm::SetTorusSize*** message is intended to set the size attributes of a ***gvm::Torus*** instance.

Parent Classes: public ***gvm::Message***

Derived Classes: None

GLdouble ***gvm::SetTorusSize::innerRadius*** – Value to set the inner radius attribute of the destination ***gvm::Torus*** instance.

GLdouble ***gvm::SetTorusSize::outerRadius*** – Value to set the outer radius attribute of the destination ***gvm::Torus*** instance.

GLint ***gvm::SetTorusSize::sides*** – Value to set the side count attribute of the destination ***gvm::Torus*** instance.

GLint ***gvm::SetTorusSize::rings*** – Value to set the ring count attribute of the destination ***gvm::Torus*** instance.

Public Constructors:

gvm::SetTorusSize::SetTorusSize**(*****gvm::View&*** *v*, **double** *t*, ***gvm::object_index*** *i*, **GLdouble** *ir*, **GLdouble** *or*, **GLint** *s*, **GLint** *r***)** – This constructor calls the parent class constructor ***gvm::Message*****(*****v*****,** ***t*****, GVM_SetTorusSize,** ***i*****)** and initializes ***innerRadius***, ***outerRadius***, ***sides***, and ***rings*** to *ir*, *or*, *s*, and *r* respectively.

Public Method:

virtual void ***gvm::SetTorusSize::send*****(void)** – This method commits the changes in the various attributes of the destination ***gvm::Torus*** instance.

B.4.35. gvm::SetTranslation

The ***gvm*::*SetTranslation*** message is intended to set the translation attribute of a ***gvm*::*Node*** instance.

Parent Classes: public *gvm*::*Message*

Derived Classes: None

Private Data Members:

std*::*vector*<GLdouble> *gvm*::*SetTranslation*::*trans – Value to set the translation attribute of the destination ***gvm*::*Node*** instance.

Public Constructors:

gvm*::*SetTranslation*::*SetTranslation*(*gvm*::*View*&** *v*, **double** *t*, ***gvm*::*object_index *i*, ***std*::*vector*<GLdouble>** *r*) – This constructor initializes ***trans*** to *r* and calls the parent class constructor ***gvm*::*Message***(*v*, *t*, **GVM_SetTranslation,** *i*).

Public Methods:

virtual void *gvm*::*SetTranslation*::*send*(void) – This method actually sets the translation attribute of the destination ***gvm*::*Node*** instance.

B.4.36. gvm::SetVertex

The ***gvm*::*SetVertex*** message is intended to set the location of the vertex associated with a ***gvm*::*Vertex*** instance.

Parent Classes: public *gvm*::*Message*

Derived Classes: None

Private Data Members:

std*::*vector*<GLdouble> *gvm*::*SetVertex*::*vert – Value to set the location attribute of the destination ***gvm*::*Vertex*** instance.

Public Constructors:

gvm::*SetVertex*::*SetVertex*(*gvm*::*View*& *v*, **double** *t*, *gvm*::*object_index* *i*, *std*::*vector*<GLdouble> *v*) – This constructor initializes ***vert*** to *v* and calls the parent class constructor ***gvm*::*Message*(*v*, *t*, GVM_SetVertex, *i*)**.

Public Methods:

virtual void *gvm*::*SetVertex*::*send*(void) – This method actually sets the location attribute of the destination ***gvm*::*Vertex*** instance.

B.4.37. gvm::Shape

This class provides basic support for displaying two and three-dimensional shapes to a GLUT window. It provides support only for rendering color, and nothing for lighting, shading or texturing.

Parent Class: public *gvm*::*Object*

Derived Classes: *gvm*::*Shape2D*, *gvm*::*Shape3D*

Protected Enumerators:

enum *gvm*::*Shape*::*shape_flags*{SF_Color, SF_PointSize, SF_LAST} – This enumerator acts as in index in the ***flags*** array.

Protected Data Members:

std*::*vector*<GLdouble> *gvm*::*Shape*::*color – Vector containing the color components of the shape instance. This is used only when ***flags*[SF_Color]** is **true**.

GLenum *gvm*::*Shape*::*mode* – Rendering mode for this shape. It takes on one of the following values **GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES,**

GL_TRIANGLE_STRIP, **GL_TRIANGLE_FAN**, **GL_QUADS**, **GL_QUAD_STRIP**, **GL_POLYGON**. The meaning of these values can be found in (Wright 1996) page 172.

GLfloat ***gvm*::*Shape*::*pt_size*** – Point size to use when rendering using **GL_POINTS**. This is only used when ***flags***[**SF_PointSize**] is **true**.

std*::*vector*<bool>** ***gvm*::*Shape*::*flags – The components of this array are used to determine whether the default values for drawing color or point size are to be used when rendering the shape.

Public Constructors:

gvm*::*Shape*::*Shape*(*gvm*::*View*& *v*, *gvm*::*object_type* *t*, *gvm*::*object_index* *i*)** – This constructor calls parent constructor ***gvm*::*Object(*v*, *t*, *i*), and constructors ***color***(4, -1.0), ***flags***(**SF_LAST, false**), and sets ***mode*** and ***pt_size*** to **GL_POINTS** and 1.0 respectively.

Public Methods:

virtual void ***gvm*::*Shape*::*begin*(void)** – Begin rendering this shape. It saves the default drawing color and point size sets the local values for them if the proper elements of ***flags*** are set. It then calls ***::glBegin*(*mode*)**.

virtual void ***gvm*::*Shape*::*end*(void)** – End rendering this shape. It calls ***::glEnd*()** and restores the default drawing color and point size in the event that the local values were used instead.

virtual bool ***gvm*::*Shape*::*isType*(*gvm*::*object_type* *t*)** – This routine returns to the calling routine **true** if *t*=**GVM_Shape** or ***gvm*::*Object*::*isType*(*t*)** returns **true**.

virtual void ***gvm*::*Shape*::*setColor*(*std*::*vector*<GLdouble> *c*)** – Sets ***color*** to ***::resize_vector***(4, 1.0, *c*). If all of the components of *c* are in the range [0, 1] then ***flags***[**SF_Color**] is set to **true**; **false** otherwise.

virtual void ***gvm*::*Shape*::*setColor*(GLdouble *r*, GLdouble *g*, GLdouble *b*)** – Sets ***color*** to ***::make_vector***(4, *r*, *g*, *b*, 1.0). If *r*, *g*, and *b* are all in the range [0, 1] then ***flags***[**SF_Color**] is set to **true**; **false** otherwise.

virtual void ***gvm*::*Shape*::*setColor*(GLdouble** *r*, **GLdouble** *g*, **GLdouble** *b*, **GLdouble** *a*) – Sets ***color*** to ***::make_vector***(4, *r*, *g*, *b*, *a*). If *r*, *g*, *b*, and *a* are all in the range [0, 1] then ***flags*[SF_Color]** is set to **true**; **false** otherwise.

virtual void ***gvm*::*Shape*::*setMode*(GLenum** *m*) – Sets ***mode*** to *m*.

virtual void ***gvm*::*Shape*::*setPointSize*(GLfloat** *ps*) – Set ***pt_size*** to *ps*.

B.4.38. gvm::Shape2D

This is the parent class for all two-dimensional shapes. It serves primarily as a placeholder for various data structures, and has no additional functionality beyond that defined in ***gvm*::*Shape***.

Parent Class: public ***gvm*::*Shape***

Derived Classes: ***gvm*::*Polygon2D***

Protected Constructors:

gvm*::*Shape2D*::*Shape2D*(*gvm*::*View2D*&** *v*, ***gvm*::*object_type *t*, ***gvm*::*object_index*** *i*) - This constructor calls the parent constructor ***gvm*::*Shape***(*v*, *t*, *i*).

Public Methods:

virtual bool ***gvm*::*Shape2D*::*isType*(*gvm*::*object_type*** *t*) – This routine returns to the calling routine **true** if *t*=**GVM_Shape2D** or ***gvm*::*Shape*::*isType***(*t*) returns **true**.

B.4.39. gvm::Shape3D

This is the parent class for all three-dimensional shapes. It serves primarily as a placeholder for various data structures, and has no additional functionality beyond ***gvm*::*Shape***.

Parent Class: public ***gvm*::*Shape***

Derived Classes: ***gvm*::*Cone***, ***gvm*::*Cube***, ***gvm*::*Cylinder***, ***gvm*::*Dodecahedron***, ***gvm*::*Icosahedron***, ***gvm*::*Octahedron***, ***gvm*::*Polygon3D***, ***gvm*::*Sphere***, ***gvm*::*Tetrahedron***, ***gvm*::*Torus***

Protected Constructors:

gvm*::*Shape3D*::*Shape3D*(*gvm*::*View3D*& *v*, *gvm*::*object_type t*, *gvm*::*object_index i*)** - This constructor calls the parent constructor ***gvm*::*Shape(*v*, *t*, *i*).

Public Methods:

virtual bool *gvm*::*Shape3D*::*isType*(*gvm*::*object_type t*) – This routine returns to the calling routine **true** if *t*=**GVM_Shape3D** or ***gvm*::*Shape*::*isType***(*t*) returns **true**.

B.4.40. gvm::Sphere

This provides a means of displaying a sphere in a GLUT window. The **::*glutSolidSphere*(*radius*, *slices*, *stacks*)** and **::*glutWireOctahedron*(*radius*, *slices*, *stacks*)** routines are called to render the sphere.

Parent Class: **public *gvm*::*Shape3D***

Derived Classes: None

Protected Data Members:

GLdouble *gvm*::*Sphere*::*radius* – Radius of the sphere.

GLint *gvm*::*Sphere*::*slices* – Number of radial slices into which to break the sphere.

GLint *gvm*::*Sphere*::*stacks* – Number of lateral stacks into which to break the sphere.

Public Constructors:

gvm*::*Sphere*::*Sphere*(*gvm*::*View3D*& *v*, ulong *i*)** – This constructor calls the parent class constructor ***gvm*::*Shape3D(*v*, **GVM_Sphere,** *i*) and initializes **radius**, **slices** and **stacks** to 1.0, 10 and 10 respectively.

Public Methods:

virtual void ***gvm*::*Sphere*::*display*(void)** – Display sphere using ***::glutSolidSphere(radius*, *slices*, *stacks)*** if ***mode*** is **GL_POLYGON** or ***::glutWireSphere(radius*, *slices*, *stacks)*** otherwise.

virtual bool ***gvm*::*Sphere*::*isType(gvm*::*object_type*** *t*) – This routine returns to the calling routine **true** if *t*=**GVM_Sphere** or ***gvm*::*Shape3D*::*isType*(*t*)** returns **true**.

virtual void ***gvm*::*Sphere*::*set*(GLdouble** *r*, **GLint** *sl*, **GLint** *st*) – Set ***radius***, ***slices*** and ***stacks*** to *r*, *sl*, and *st*.

virtual void ***gvm*::*Sphere*::*setRadius*(GLdouble** *r*) – Set ***radius*** to *r*.

virtual void ***gvm*::*Sphere*::*setSlices*(GLint** *s*) – Set ***slices*** to *s*.

virtual void ***gvm*::*Sphere*::*setStacks*(GLint** *s*) – Set ***stacks*** to *s*.

B.4.41. gvm::Tetrahedron

This provides a means of displaying a tetrahedron in a GLUT window. The ***::glutSolidTetrahedron*()** and ***::glutWireTetrahedron*()** routines are called to render the tetrahedron.

Parent Class: **public** ***gvm*::*Shape3D***

Derived Classes: None

Public Constructors:

gvm*::*Tetrahedron*::*Tetrahedron(gvm*::*View3D& *v*, **ulong** *i*) – This class constructor calls the parent constructor ***gvm*::*Shape3D*(*v*, GVM_Tetrahedron,** *i*).

Public Methods:

virtual void ***gvm*::*Tetrahedron*::*display*(void)** – Display a tetrahedron using ***::glutSolidTetrahedron*()** if ***mode*** is **GL_POLYGON** or ***::glutWireTetrahedron*()** otherwise.

virtual bool ***gvm*::*Tetrahedron*::*isType*(*gvm*::*object_type* t)** – This routine returns to the calling routine **true** if *t*=**GVM_Tetrahedron** or ***gvm*::*Shape3D*::*isType*(*t*)** returns **true**.

B.4.42. gvm::Torus

The ***gvm*::*Torus*** class provides a means of displaying a torus in a GLUT window. The ***::glutSolidTorus*(*innerRadius*, *outerRadius*, *nsides*, *rings*)** and ***::glutWireTorus*(*inner*, *outer*, *nsides*, *rings*)** routines are called to render the torus.

Parent Class: **public *gvm*::*Shape3D***

Derived Classes: None

Protected Data Members:

GLdouble *gvm*::*Torus*::*innerRadius* – Inner radius of the torus.

GLint *gvm*::*Torus*::*nsides* – Number of sides around a ring.

GLdouble *gvm*::*Torus*::*outerRadius* – Outer radius of the torus.

GLint *gvm*::*Torus*::*rings* – Number of rings around the torus.

Public Constructors:

gvm*::*Torus*::*Torus*(*gvm*::*View3D*& *v*, ulong *i*)** – This constructor calls the parent class constructor ***gvm*::*Shape3D*(*v*, GVM_Torus, *i*)** and initializes ***innerRadius, ***outerRadius***, ***nsides*** and ***rings*** to 1.0, 2.0, 10 and 10 respectively.

Public Methods:

virtual void *gvm*::*Torus*::*display*(void) – Display a torus using either of the routines ***::glutSolidTorus*(*innerRadius*, *outerRadius*, *nsides*, *rings*)** if ***mode*** is **GL_POLYGON** or ***::glutWireTorus*(*innerRadius*, *outerRadius*, *nsides*, *rings*)** otherwise.

virtual bool ***gvm*::*Torus*::*isType*(*gvm*::*object_type*** *t*) – This routine returns to the calling routine **true** if *t*=**GVM_Torus** or ***gvm*::*Shape3D*::*isType***(*t*) returns **true**.

virtual void ***gvm*::*Torus*::*set***(**GLdouble** *i*, **GLdouble** *o*, **GLint** *n*, **GLint** *r*) – Sets ***innerRadius***, ***outerRadius***, ***nsides*** and ***rings*** to *i*, *o*, *n* and *r* respectively.

virtual void ***gvm*::*Torus*::*setInnerRadius***(**GLdouble** *i*) – Sets ***innerRadius*** to *i*.

virtual void ***gvm*::*Torus*::*setNSides***(**GLint** *n*) – Sets ***nsides*** to *n*.

virtual void ***gvm*::*Torus*::*setOuterRadius***(**GLdouble** *o*) – Sets ***outerRadius*** to *o*.

virtual void ***gvm*::*Torus*::*setRings***(**GLint** *r*) – Sets ***rings*** to *r*.

B.4.43. gvm::Vertex

gvm*::*Vertex provides the basic functionality for generic vertices. Derived classes deal with vertices in specific vector spaces, notably 2 and 3-spaces.

Parent Class: **public** ***gvm*::*Object***

Derived Classes: ***gvm*::*Vertex2D***, ***gvm*::*Vertex3D***

Protected Data Members:

std*::*vector*<GLdouble>** ***gvm*::*Vertex*::*loc – Location of the vertex.

Protected Constructors:

gvm*::*Vertex*::*Vertex*(*gvm*::*View& *v*, ***gvm*::*object_type*** *t*, ***gvm*::*object_index*** *i*) – This constructor calls the parent constructor ***gvm*::*Object***(*v*, *t*, *i*) and initializes ***loc*** to either <0.0, 0.0> in the case ***this** is a ***gvm*::*Vertex2D*** instance or <0.0, 0.0, 0.0> when it is a ***gvm*::*Vertex3D***.

Public Methods:

virtual void ***gvm*****::*****Vertex*****::*****set*****(const** ***std*****::*****vector*****<GLdouble>&** *v*) – This method sets ***loc*** to **::*****resize_vector*****(*****loc.size*****()**, 0.0, *v*).

virtual bool ***gvm*****::*****Vertex*****::*****isType*****(*****gvm*****::*****object_type*** *t*) – This routine returns to the calling routine **true** if *t*=**GVM_Vertex** or ***gvm*****::*****Object*****::*****isType*****(*t*) returns **true**.

B.4.44. gvm::Vertex2D

gvm**::*****Vertex2D*** provides specializes ***gvm*****::*****Vertex*** to perform two-dimensional vertex operations.

Parent Class: **public** ***gvm*****::*****Vertex***

Derived Classes: None

Public Constructors:

gvm**::*****Vertex2D*****::*****Vertex2D*****(*****gvm*****::*****View2D*****&** *v*, **ulong** *i*) – This constructor calls the parent class constructor ***gvm*****::*****Vertex*****(*v*, **GVM_Vertex2D**, *i*).

Public Methods:

virtual void ***gvm*****::*****Vertex2D*****::*****display*****(void)** – This routine displays the vertex at its specified location, < ***gvm*****::*****Vertex*****::*****loc***[0], ***gvm*****::*****Vertex*****::*****loc***[1]>

virtual bool ***gvm*****::*****Vertex2D*****::*****isType*****(*****gvm*****::*****object_type*** *t*) – This routine returns to the calling routine **true** if *t*=**GVM_Vertex2D** or ***gvm*****::*****Vertex*****::*****isType*****(*t*) returns **true**.

virtual void ***gvm*****::*****Vertex2D*****::*****set*****(GLdouble** *x*, **GLdouble** *y*) – This method sets the vector ***gvm*****::*****Vertex*****::*****loc*** to **::*****make_vector*****(2,** *x*, *y*).

B.4.45. gvm::Vertex3D

gvm**::*****Vertex3D*** provides specializes ***gvm*****::*****Vertex*** to perform three-dimensional vertex operations.

Parent Class: **public** ***gvm*****::*****Vertex***

Derived Classes: None

Public Constructors:

***gvm*::*Vertex3D*::*Vertex3D*(*gvm*::*View3D*& *v*, ulong *i*)** – This constructor calls the parent class constructor ***gvm*::*Vertex*(*v*, GVM_Vertex3D, *i*)**.

Public Methods:

virtual void *gvm*::*Vertex3D*::*display*(void) – This routine displays the vertex at its specified location, < ***gvm*::*Vertex*::*loc***[0] , ***gvm*::*Vertex*::*loc***[1] , ***gvm*::*Vertex*::*loc***[2]>

virtual bool *gvm*::*Vertex3D*::*isType*(*gvm*::*object_type t*) – This routine returns to the calling routine **true** if *t*=**GVM_Vertex3D** or ***gvm*::*Vertex*::*isType***(*t*) returns **true**.

virtual void *gvm*::*Vertex3D*::*set*(GLdouble *x*, GLdouble *y*, GLdouble *z*) – This method sets the vector ***gvm*::*Vertex*::*loc*** to ***::make_vector*(3, *x*, *y*, *z*)**.

B.4.46. gvm::View

The ***gvm*::*View*** class is the parent for the two classes specifically intended for two and three dimensional graphical output, ***gvm*::*View2D*** and ***gvm*::*View3D*** respectively.

Parent Classes: **public *sodl*::*Trace***

Derived Classes: ***gvm*::*View2D*, *gvm*::*View3D***

Protected Data Members:

float *gvm*::*View*::*aspect* – This is the current aspect ratio of the output display. It is updated any time there is a change in the size of the viewport window.

std*::*vector*<bool> *gvm*::*View*::*buttonState – This vector is used to keep track of the mouse button states. When button *xxx* is pressed, ***buttonState*[GLUT_*xxx*_BUTTON]** is true, where *xxx* is one of {**LEFT, MIDDLE, RIGHT**}.

bool ***gvm::View::isVisible*** – This value is **true** when the window is visible, **false** otherwise.

std::vector*<uint>** ***gvm::View::mouseLoc – This vector is used to track the current mouse position. When the mouse is at GLUT window location <*x*, *y*>, then ***mouseLoc***[0]=*x* and ***mouseLoc***[1]=*y*.

std::list<gvm::Message*> ***gvm::View::msgList*** – This is the pending message list. It is always sorted by time stamp. Messages with earlier timestamps are at the front of the list, and later ones are at the back.

ulong ***gvm::View::nextMessage*** – This is a counter for the number of messages that have been created for the view. Each new message is given a unique handle, part of which is the instance number it derived from the value of the ***nextMessage*** data member. As new messages are added, the ***nextMessage*** field is incremented.

gvm::object_index ***gvm::View::nextObject*** – This is a counter for the number of objects that have been created for the view. Each new object is given a unique handle, part of which is the instance number it derived from the value of the ***nextObject*** data member. As new objects are added, the ***nextObject*** field is incremented.

std::vector<gvm::Object*> ***gvm::View::nodeList*** – This is the list of root nodes for the ***gvm::View*** instance. These are polled during the display phase, and each one is displayed in turn.

std::vector<gvm::Object*> ***gvm::View::objectList*** – This is the master list of all of the objects in the view. As each new object *obj* is created, a pointer to it occupies ***objectList***[***obj.getIndex***()].

GLfloat ***gvm::View::ptSize*** – This is the view's default point size. Any subordinate objects in the view's scene graph that do not explicitly override the point size will use this default value.

bool ***gvm::View::refresh*** – This flag is set to **true** when there has been a request to refresh the display. The actual screen refresh occurs during the fossil collection phase.

bool ***gvm::View::sceneChange*** – This is set to **true** when some component of the scene has changed its state.

std::string gvm::View::time – This is the time stamp of the current scene.

std::string gvm::View::title – This acts as a title for the GLUT window containing the scene.

ulong ***gvm::View::window*** – This is the GLUT window number for this view. When a GLUT window is created, it is assigned a unique identifier, which we retain in ***window***.

std::vector**<bool>** ***gvm::View::zoom*** – This is a flag for determining whether or not to zoom into or away from a scene. When the '+' or '=' keys are pressed, ***zoom***[0] is set to **true**. When '-' or '_' is pressed, ***zoom***[1] is set to **true**.

Protected Constructors:

gvm::View::View**(void)** – This is the default class constructor for the ***gvm::View*** class. It initializes ***nextObject***, ***nextMessage***, ***ptSize***, ***sceneChange*** and ***refresh*** to 0, 0, 1.0, **false** and **true** respectively. It also calls the initializers ***buttonState***(3, **false**), ***mouseLoc***(2, 0) and ***window(glutCreateWindow***("sodl Display")).

Public Methods:

virtual void ***gvm::View::addNode(gvm::object_index i)*** – This routine adds to the back of the ***nodeList*** array, ***objectList***[*i*].

virtual void ***gvm::View::begin*****(void)=0** – This abstract method is used to set up any view-dependent settings related to view position, orientation, and the like.

virtual ***gvm::object_index gvm::View::createObject*****(double** *t*, ***gvm::object_type*** *type*) – This method scheduled the creation of a new object of type *type* for time *t*.

virtual void ***gvm::View::createObject(gvm::object_handle h)*****=0** – This abstract method is meant to have derived classes actually create an object with the handle specified. Once created, a pointer to it is inserted at ***objectList***[h.***first***].

virtual void ***gvm::View::display*****(void)** – If ***isDisplay***() returns true, then this method will call the ***begin*** method, set the default point size and perform some boiler-plate OpenGL routines. After that, it calls the ***display*** method for each of the ***gvm::Node*** instance pointed to in the ***nodeList***. Afterwards, the ***end*** method is called. Normally, the GLUT run-time system will inform the controlling ***sodl::GLUTViewManager*** instance that a display update needs to occur, which in turn calls this method. It is unwise to directly call the method, since there is some additional processing that occurs outside of the method that needs to be performed first. Requests for a redisplay of the screen can be made by a call to ***::glutPostRedisplay***().

virtual void ***gvm::View::end*****(void)** – This method performs some cleanup after the scene graph is displayed.

virtual void ***gvm::View::entry*****(int** *e***)** – This method is called from with the controlling ***sodl::GLUTViewManager*** whenever the mouse either enters or leaves the window associated with this view. Parameter *e* takes on either of the values **GLUT_ENTERED** if the mouse curser entered the window, or **GLUT_LEFT** if the mouse left the window.

virtual void ***gvm::View::fossilCollect*****(double** *t***)** – The owning **process:*View*** instance calls this method during incremental fossil collection to time *t*. This fossil collection routine updates the scene graph, processing any messages with a time stamp equal to *t*. There should be no messages with a time stamp less then *t*, since those messages should have been processed during an earlier fossil collection cycle.

virtual ***std::string gvm::View::getTime*****(void) const** – This routine returns ***time*** to the calling routine

virtual ***std::string gvm::View::getTitle*****(void) const** – This routine returns ***title*** to the calling routine

virtual ulong ***gvm::View::getWindow*****(void) const** – This routine returns ***window*** to the calling routine.

virtual bool ***gvm::View::isDisplay*****(void)=0** – This abstract method returns ***true*** exactly when the derived class instance actually implementing this method has determined that it needs to redraw its scene graph.

virtual void ***gvm::View::keydown*****(byte** *key*, **int** *x*, **int** *y*) – When the GLUT run-time system detects a key press event for the GLUT window associated with this ***gvm::View*** instance, the controlling

sodl::GLUTViewManager is notified, which in turn calls this method to notify the view. Parameter *key* is the ASCII value of the key that was pressed, and the mouse position at the time of the key press is given by <*x*, *y*>.

virtual void ***gvm::View::keyup*****(byte** *key*, **int** *x*, **int** *y*) – When the GLUT run-time system detects a key release event for the GLUT window associated with this ***gvm::View*** instance, the controlling ***sodl::GLUTViewManager*** is notified, which in turn calls this method to notify the view. Parameter *key* is the ASCII value of the key that was released, and the mouse position at the time of the key press is given by <*x*, *y*>.

virtual void ***gvm::View::motion*****(int** *x*, **int** *y*) – When the GLUT run-time system detects an active mouse motion event (i.e. one where at least one mouse button is pressed while the mouse is moved) in the GLUT window associated with this ***gvm::View*** instance, the controlling ***sodl::GLUTViewManager*** is notified. This in turn calls this method to notify the view. The position of the mouse is given by <x, y>.

virtual void ***gvm::View::mouse*****(int** *button*, **int** *state*, **int** *x*, **int** *y*) – When the GLUT run-time system detects an active mouse button event (i.e. one where at least one mouse button changes its state) in the GLUT window associated with this ***gvm::View*** instance, the controlling ***sodl::GLUTViewManager*** is notified. This in turn calls this method to notify the view. Parameter *button* refers to the button that was actually had the event: **GLUT_LEFT_BUTTON** for the left mouse button, **GLUT_MIDDLE_BUTTON** for the middle button or **GLUT_RIGHT_BUTTON** for the right button. The *state* parameter describes the new mouse button position: **GLUT_UP** for a button release; **GLUT_DOWN** for a button press. The position of the mouse at the time of the button event is given by <x, y>.

virtual ***gvm::Object& gvm::View::operator[](gvm::object_index*** *i*) – This routine returns ****objectList***[*i*] to the calling routine. In the event that *i* is out of range, this routine will throw an ***Exception::RangeError***.

virtual void ***gvm::View::overlay*****(void)** – When the GLUT run-time system detects an overlay event in the GLUT window associated with this ***gvm::View*** instance, the controlling ***sodl::GLUTViewManager*** is notified, which in turn calls this method to notify the view.

virtual void ***gvm*::*View*::*passive_motion*(int** *x*, **int** *y*) – When the GLUT run-time system detects a passive mouse motion event (i.e. one where no mouse button is pressed while the mouse is moved) in the GLUT window associated with this ***gvm*::*View*** instance, the controlling ***sodl*::*GLUTViewManager*** is notified, which in turn calls this method to notify the view. The position of the mouse is given by <*x*, *y*>.

virtual void ***gvm*::*View*::*resdisplay*(void)** – This method requests that the GLUT engine redisplay the scene associated with this view by calling ***glutSetWindow*(*window*)** followed by ***glutPostRedisplay*()**.

virtual void ***gvm*::*View*::*reshape*(int** *width*, **int** *height*) – When the GLUT run-time system detects a reshape event to size <*width*, *height*> in the GLUT window associated with this ***gvm*::*View*** instance, the controlling ***sodl*::*GLUTViewManager*** is notified, which in turn calls this method to notify the view. In this case, OpenGL needs to be informed of the change in the viewport parameters, and ***aspect*** needs to be updated to reflect the new window aspect ratio.

virtual void ***gvm*::*View*::*restore*(double** *t*) – When the owning **process:*View*** instance receives a rollback request to time *t*, it calls this method. Since the new *t* is the new time stamp for this ***gvm*::*View*** instance, all message previously scheduled for times at or after t are now invalid, and must be removed from ***messageList***. This method accomplishes this removing them from the front of ***messageList***.

virtual void ***gvm*::*View*::*schedule*(*gvm*::*Message**** *msg*) – Messages are scheduled for later processing with this method. The new message inserted at the back of ***msgList***. By virtue of the Time Warp algorithm, (**msg*).***getTime*()** is never less than (****messageList.back*()).*getTime*()**. Thus, the messages appear in ***msgList*** in ascending time stamp order from the earliest at the front to the latest in the back.

virtual void ***gvm*::*View*::*setPointSize*(GLfloat** *ps*) – This routine sets ***ptSize*** to *ps*.

virtual void ***gvm*::*View*::*setPosition*(int** *x*, **int** *y*) – This routine sets the position of the GLUT window to <*x*, *y*>.

virtual void ***gvm*::*View*::*setRefresh*(bool** *r*) – This routine sets the ***refresh*** flag to *r*.

virtual void ***gvm*::*View*::*setSceneChange*(bool** *s*) – This routine sets the ***sceneChange*** flag to *s*.

virtual void ***gvm*****::*****View*****::*****setSize*****(int** *w*, **int** *h*) – This routine sets the size of the GLUT window to <*w*, *h*>. In this case, the GLUT run-time system will notify the ***sodl*****::*****GLUTViewManager*** controlling this view of the change, which will in turn call the ***reshape*** method described above.

virtual void ***gvm*****::*****View*****::*****setTitle*****(*****std*****::*****string*** *str*) – This routine will set ***title*** to *str* and reset the GLUT window and icon titles to *str*.

virtual void ***gvm*****::*****View*****::*****specialdown*****(int** *key*, **int** *x*, **int** *y*) – When the GLUT run-time system detects a special key press events for the GLUT window associated with this ***gvm*****::*****View*** instance, the controlling ***sodl*****::*****GLUTViewManager*** is notified, which in turn calls this method to notify the view. Parameter *key* takes on one of the following values: **GLUT_KEY_F1**, **GLUT_KEY_F2**, **GLUT_KEY_F3**, **GLUT_KEY_F4**, **GLUT_KEY_F5**, **GLUT_KEY_F6**, **GLUT_KEY_F7**, **GLUT_KEY_F8**, **GLUT_KEY_F9**, **GLUT_KEY_F10**, **GLUT_KEY_F11**, **GLUT_KEY_F12**, **GLUT_KEY_LEFT**, **GLUT_KEY_UP**, **GLUT_KEY_RIGHT**, **GLUT_KEY_DOWN**, **GLUT_KEY_PAGE_UP**, **GLUT_KEY_PAGE_DOWN**, **GLUT_KEY_HOME**, **GLUT_KEY_END**, or **GLUT_KEY_INSERT**. The mouse position at the time of the key press is given by <*x*, *y*>.

virtual void ***gvm*****::*****View*****::*****special*****(int** *key*, **int** *x*, **int** *y*) – When the GLUT run-time system detects a special key release event for the GLUT window associated with this ***gvm*****::*****View*** instance, the controlling ***sodl*****::*****GLUTViewManager*** is notified, which in turn calls this method to notify the view. Parameter *key* takes on one of the listed in the description of ***specialdown***. The mouse position at the time of the key release is given by <*x*, *y*>.

virtual void ***gvm*****::*****View*****::*****visible*****(int** *vis*) – When the GLUT run-time system detects a visibility event (i.e. one in which the window either becomes visible or hidden) for the GLUT window associated with this ***gvm*****::*****View*** instance, the controlling ***sodl*****::*****GLUTViewManager*** is notified, which in turn calls this method to notify the view. Parameter *vis* takes on one of the values **GLUT_NOT_VISIBLE** or **GLUT_VISIBLE**.

B.4.47. gvm::View2D

The ***gvm::View2D*** class expands somewhat the functionality of the ***gvm::View*** class above. This extension is mainly in the form of allowing only certain types of objects to be added to ***gvm::View::objectList***, and how it handles input events from the user.

Parent Classes: **public** ***gvm::View***

Derived Classes: None

Private Data Members:

float ***gvm::View2D::transX*** – This holds the translation factor in the X direction for viewing the scene.

float ***gvm::View2D::transY*** – This holds the translation factor in the Y direction for viewing the scene.

float ***gvm::View2D::sdepth*** – This is used to position the view at different distances "above" the view, allowing it to be uniformly scaled.

float ***gvm::View2D::zNear*** – This is the near clipping plane.

float ***gvm::View2D::zFar*** – This is the far clipping plane

Public Constructors:

gvm::View2D::View2D**(void)** – This class constructor initializes ***transX***, ***transY***, ***sdepth***, ***zNear***, ***zFar*** to 0.0, 0.0, 80.0, 1.0 and 1000.0 respectively.

Public Methods:

virtual void ***gvm::View2D::addNode(gvm::object_index n)*** – This routine will add the ***gvm::Node2D*** instance with index *n* to the back of ***gvm::View::nodeList***.

virtual void ***gvm::View2D::begin*****(void)** – This method performs some scene initialization that is useful in specifying how to display the scene graph. Among other things, it positions the viewer in a useful location.

In addition, if ***zoom***[0] is set, the view zooms into the scene, and when ***zoom***[1] is set, it zooms away from the scene.

virtual void ***gvm::View2D::createObject*** (***gvm::object_handle h***) – This method will create a ***gvm::Object*** instance of type ***h.first*** and with index ***h.second***. Object types are limited to **GVM_Node2D**, **GVM_Polygon2D**, and **GVM_Vertex2D**. Once created, a pointer to the new object is entered into ***gvm::View::objectList***[***h.second***].

virtual void ***gvm::View2D::end***(**void**) – This method performs some cleanup actions in the view, notably changing the display buffers to display what had just been drawn.

virtual bool ***gvm::View2D::isDisplay***(**void**) – This method returns **true** exactly when the ***isVisible***, ***refresh***, and ***sceneChange*** methods return **true**. This will enable the scene to be updated on the screen only when there is a change in the scene graph and when there has been a request to change the display.

virtual void ***gvm::View2D::motion***(**int** *x*, **int** *y*) – This method changes the view parameters to allow the user to interact with the scene. When the left mouse button is pressed, the scene is proportionally translated in plane with the mouse motion. When the middle button is pressed, the scene is scaled proportionally with the y-displacement of the mouse.

B.4.48. gvm::View3D

The ***gvm::View3D*** class expands somewhat the functionality of the ***gvm::View*** class above. This extension is mainly in the form of allowing only certain types of objects to be added to ***gvm::View::objectList***, and how it handles input events from the user.

Parent Classes: **public** ***gvm::View***

Derived Classes: None

Private Data Members:

std::valarray< **GLdouble** > ***gvm::View3D::pos*** – Position of the view point.

std*::*valarray*< GLdouble > *gvm*::*View3D*::*ori – Orientation of the view.

GLdouble *gvm*::*View3D*::*scale* – Zooming factor of the scale.

GLdouble *gvm*::*View3D*::*zNear* – Near clipping plane.

GLdouble *gvm*::*View3D*::*zFar* – Far clipping plane.

Public Constructors:

gvm*::*View3D*::*View3D*(void)** – This constructor initializes ***pos, ***ori***, ***scale***, ***zNear*** and ***zFar*** to <0.0, 0.0, -80.0>, <90.0, 0.0, -45.0>, 1.0, 1.0 and 1000.0 respectively.

Public Methods:

virtual void *gvm*::*View3D*::*addNode*(*gvm*::*object_index i*) – This routine will add the ***gvm*::*Node3D*** instance with index *n* to the back of ***gvm*::*View*::*nodeList***.

virtual void *gvm*::*View3D*::*begin*(void) – This method performs some scene initialization that is useful in specifying how to display the scene graph. Among other things, it positions the viewer in a useful location. In addition, if ***zoom***[0] is set, the view zooms into the scene, and when ***zoom***[1] is set, it zooms away from the scene.

virtual void *gvm*::*View3D*::*createObject*(*gvm*::*object_handle h*) – This method will create a ***gvm*::*Object*** instance of type ***h.first*** and with index ***h.second***. Object types are limited to **GVM_Cone**, **GVM_Cube**, **GVM_Cylinder**, **GVM_Dodecahedron**, **GVM_Icosahedron**, **GVM_Node3D**, **GVM_Octahedron**, **GVM_Polygon3D**, **GVM_Sphere**, **GVM_Tetrahedron**, **GVM_Torus** and **GVM_Vertex3D**. Once created, a pointer to the new object is entered into ***gvm*::*View*::*objectList*[*h.second*]**.

virtual void *gvm*::*View3D*::*display*(void) – This method performs some setup for displaying the 3D information in the GLUT window associated with this instance. If ***zoom***[0] is set, the view zooms into the scene, and when ***zoom***[1] is set, it zooms away from the scene.

virtual void ***gvm*::*View3D*::*end*(void)** – This method performs some cleanup actions in the view, notably changing the display buffers to display what had just been drawn.

virtual bool ***gvm*::*View3D*::*isDisplay*(void)** – This method returns ***true*** exactly when the ***gvm*::*View*::*isVisible***, ***gvm*::*View*::*refresh***, and ***gvm*::*View*::*sceneChange*** methods return true. This will enable the scene to be updated on the screen only when there is a change in the scene graph and when there has been a request to change the display.

virtual void ***gvm*::*View3D*::*motion*(int** *x*, **int** *y*) – This method is called when an active mouse event occurs within the GLUT window associated with this ***gvm*::*View3D*** instance. When the left mouse button is pressed, the scene rotates about the screen x-axis proportional to mouse displacement in the y-direction and rotation about the y-axis is performed proportionally to mouse displacement in the x-direction. When the middle button is pressed, the scene is scaled proportionally with the y-displacement of the mouse.

B.4.49. GVM Definitions not associated with a specific class

Enumerators:

enum ***gvm*::*message_type*{GVM_AddNode, GVM_AddShape, GVM_AddVertex, GVM_CreateObject, GVM_Refresh, GVM_SetActive, GVM_SetColor, GVM_SetConeSize, GVM_SetCubeSize, GVM_SetCylinderSize, GVM_SetLabel, GVM_SetMode, GVM_SetPointSize, GVM_SetPosition, GVM_SetRotation, GVM_SetRotationCenter, GVM_SetScale, GVM_SetScaleCenter, GVM_SetSize, GVM_SetSphereSize, GVM_SetTorusSize, GVM_SetTranslation, GVM_SetVertex, ..., GVM_LAST_MESSAGE}** – Specifies the various message types associated with the GVM. There are a number of user messages that can be used for figures not defined here. These type labels take on the form GVM_UserMessage000, ... GVM_UserMessage255.

enum ***gvm*::*object_type*{GVM_Cone, GVM_Cube, GVM_Cylinder, GVM_Dodecahedron, GVM_Icosahedron, GVM_Node, GVM_Node2D, GVM_Node3D, GVM_Object, GVM_Octahedron, GVM_Polygon2D, GVM_Polygon3D, GVM_Shape, GVM_Shape2D, GVM_Shape3D, GVM_Sphere, GVM_Tetrahedron, GVM_Torus, GVM_Vertex, GVM_Vertex2D, GVM_Vertex3D, ...,**

GVM_LAST_OBJECT} – Specifies the various object types associated with the GVM. User defined processes can make use of pre-specified labels. These type labels take on the form GVM_UserProcess000, … GVM_UserProcess255.

Type Definitions:

typedef unsigned long ***gvm::object_index*** – Unique identifier for a ***gvm::Object*** instance assigned to a specific ***gvm::View*** instance.

typedef ***std::pair<gvm::object_type, gvm::object_index> gvm::object_handle*** – A compact form for specifying both a ***gvm::Object*** instance's type and identifier.

Functions:

std::string gvm::typeName(gvm::message_type *t*) – Returns a string representation of the message type *t*.

std::ostream& gvm::operator<<(std::ostream& *os*, **const** ***gvm::message_type*** *t*) – Sends the string representation of *t* to *os*.

std::string gvm::typeName(gvm::object_type *t*) – Returns a string representation of the object type *t*.

std::ostream& gvm::operator<<(std::ostream& *os*, **const** ***gvm::object_type*** *t*) – Sends the string representation of *t* to *os*.

Appendix C. Sample Code Listings

These code samples those used to produce the demonstrations packaged with the SODL distribution. In some cases they have been modified slightly to remove debugging code and for formatting purposes. In all cases, the essential functionality of the code has been maintained.

C.1. Battle

C.1.1. Add Environment.msg

```
{import message {SetValue} }
{message:AddEnvironment(SetValue);}
```

C.1.2. AddTrack.msg

```
{import message {TrackMotionEvent} }
{message:AddTrack(TrackMotionEvent);}
```

C.1.3. AdjustFormation.msg

```
{import message {MoveTo} }
{import gvm {gvmTank} }

/*
    This message is used to set the structure, position, orientation and
    spacing of a platoon formation.  The platoon will form up on the lead tank
    which will be in the center of the formation, and will be at the location
    specified in pos.  All of the tanks will face direction ori with distance
    between adjacent tanks norm(ori).  The left flank will be spread along
    a line emanating at angle left to the left of the lead tank.  When left is
    0.0, the left side of the formation will be abreast the lead tank.
    Positive values of left will rotate the line forward; negative values will
    rotate the line back.  The same case holds for the right flank.

    There are some predefined formations:
        LINE_ABREAST  => left = 0.0, right = 0.0
        V_FORMATION   => left = -PI/8, right = -PI/8
        FORWARD_SWEEP => left = PI/8, right = PI/8
        COLUMN        => left = PI/2, right = -PI/2
*/

{
  message:AdjustFormation(MoveTo)
  {                                           // message:AdjustFormation(MoveTo)
    double:left(0.0);                                        // Left flank angle
    double:right(0.0);                                      // Right flank angle

    method:setOri(public; void; double:x; double:y;) // Orientation vector vals
      { ori[0]=x; ori[1]=y; ori[2]=0.0; }       // Set the orientation valarray
    method:setOri(public; void; double:x; double:y; double:z;)   // Orientation
      { ori[0]=x; ori[1]=y; ori[2]=z; }         // Set the orientation valarray

    method:setFlank(public; void; double:l; double:r;)
         { left=l; right=r; }                   // Set the formation parameters
    method:getLeft(public; double;) { return left; }     // Get left flank angle

    method:getRight(public; double;) { return right; } // Get right flank angle

    method:setForm(public; void; ulong:form;)       // Use a standard formation
    {
      switch (form)
      {
        case LINE_ABREAST: left = right = 0.0; break;
        case V_FORMATION: left = right = -PI/8.0; break;
        case FORWARD_SWEEP: left = right = PI/8.0; break;
        case COLUMN: left = PI/2.0; right = -PI/2.0; break;
        default: left=right=0.0;                        // Default is line abreast
      }
    }
  }                                         // message:AdjustFormation(MoveTo)
}
```

C.1.4. Attack.msg

```
{
  message:Attack
  {
    ulong:track;                                               // Track to attack
    method:set(public; void; ulong:t;) { track=t; }       // Set the track index
    method:get(public; ulong;) { return track; }          // Get the track index
  }                                                           // message:Attack
}
```

C.1.5. Battle.proc

```
{import process {BattleView, Node3D, RedCompany, BlueCompany, Ground,
                 Environment} }
{import message {AddNode3D, AddShape3D, StartSimulation, SetFormation,
                 SetEnvironment, SetRefresh} }
{import spt {sptEnvironmentObject, sptNewtonianMotion, sptLinearMotion,
             sptAngularMotion} }

{debug false}

{
  process:Battle
  {                                                        // process:Battle
    BattleView:view;                              // Main view for the battle
    Node3D:root;                                   // Root node for the view
    Ground:ground;                                           // Play region
    BlueCompany:blue;                                         // Blue force
    RedCompany:red;                                            // Red force
    Environment:environment;          // Environment in which simulation exists

    mode:Default
    {                                                        // mode:Default
      node:start_sim[StartSimulation:strt]            // Bootstrapping message
                    [AddNode3D:an=>(view;),       // Add a node to the main view
                     AddShape3D:as=>(root;),  // Add the shape to the root node
                     SetEnvironment:se=>(blue; red; ground;),
                     SetRefresh:sr=>(view; red; blue;)]
      {                         // node:start_sim[StartSimulation][ ... ]
        EngineStand::stand.addHold(1.0);              // Set a hold at time 0.0
        an.add(root);                          // Add the root node to the view
        as.add(ground);                          // Add the ground to the node
        se.setNode(root);                    // Tell the forces of the root node
        se.setEnvironment(environment);                  // Set the environment
        sr.set(0.5);                                           // Refresh rate
      }                         // node:start_sim[StartSimulation][ ... ]
    }                                                        // mode:Default
  }                                                        // process:Battle
}
```

C.1.6. BattleView.proc

```
{import process {View3D, Tank, CommandPost, NewtonianMotion, Munition,
                 Ground} }
{import message {SetTankState, SetNewtonianMotion, Destroyed, Explosion} }
{import gvm {gvmBattleView, gvmNewtonianMotion, gvmCommandPost, gvmTank,
             gvmSetTankState, gvmMunition, gvmGround, gvmSetNewtonianMotion,
             gvmGrid, gvmSetActive, gvmExplosion} }
```

```
{debug false}

{
  process:BattleView(View3D)
  {                                                // process:BattleView(View3D)
    method:init(public; void;)
    {                                              // method:init(public; void;)
      view = new gvm::BattleView;                              // Create a new view
      View::init();                           // Call the parent class initializer
      view->setSize(1020,1063);                        // Set the size of the window
    }                                              // method:init(public; void;)

    method:getGVMType(protected; gvm::object_type; ptype:t;)
    {                  // method:getGVMType(protected; gvm::object_type; ptype;)
      switch(t)                                                // Which one is it?
      {
        case SPT_Tank: return gvm::GVM_Tank;
        case SPT_CommandPost: return gvm::GVM_CommandPost;
        case SPT_Munition: return gvm::GVM_Munition;
        case SPT_Ground: return gvm::GVM_Ground;
        default: return View3D::getGVMType(t);
      }                                                               // switch(t)
    }                  // method:getGVMType(protected; gvm::object_type; ptype;)

    mode:Default
    {                                                            // mode:Default
      node:setTankState[SetTankState:in][]
      {                                  // node:setTankState[SetTankState:in][]
        view->schedule(new gvm::SetTankState(*view,getTime(),in.index,
                                   in.az,in.azRate,in.azStart,in.azStop,
                                   in.el,in.elRate,in.elStart,in.elStop));
      }                                  // node:setTankState[SetTankState:in][]

      node:setNewtonianMotion[SetNewtonianMotion:in][]
      {                      // node:setNewtonianMotion[SetNewtonianMotion:in][]
        view->schedule(new gvm::SetNewtonianMotion(*view,getTime(),
                           in.index, in.get()));
      }                      // node:setNewtonianMotion[SetNewtonianMotion:in][]

      node:destroyed[Destroyed:in][]
      {                                        // node:destroyed[Destroyed:in][]
        view->schedule(new gvm::SetActive(*view, getTime(), in.index, false));
      }                                        // node:destroyed[Destroyed:in][]

      node:explosion[Explosion:in][]
      {                                        // node:destroyed[Destroyed:in][]
        view->schedule(new gvm::Explosion(*view,getTime(),in.index,in.get()));
      }                                        // node:destroyed[Destroyed:in][]
    }                                                            // mode:Default
  }                                                // process:BattleView(View3D)
}
```

C.1.7. BlueCompany.proc

```
{import process {Company} }
{import message {StartSimulation, SetColor, SetFormation, SetLinearPosition,
                 MoveFormation} }
{import spt {sptEnvironmentObject} }
{import gvm {gvmTank} }
{import {<math.h>} }


{
  process:BlueCompany(Company)
  {                                              // process:BlueCompany(Company)
    method:init(public; void;)
    {                                              // method:init(public; void;)
      Company::init();                // Initialize the parent construct first
      view->setTitle("Blue Tactical View");                   // Blue view title
      force = BLUE;                         // This has force designator "BLUE"
      view->setPosition(1032,574);          // Set the position of the window
      view->setSize(563, 516);                                     // Set the size
    }                                                    // method:init(public; void;)

    mode:startup
    {                                                                // mode:startup
      node:start[StartSimulation:in]                               // Upon startup
                [SetColor:sc=>(platoons; cp;),  // Set the color of the objects
                 SetFormation:sf[]=>(platoons[@];),         // Platoon formation
                 SetLinearPosition:slp=>(cp;),         // Command post position
                 MoveFormation:smove=>(platoons[3];):(10.0),
                 MoveFormation:mf[]=>(platoons[@];):(@<3 ? 10.0 : 10.0+@*5.0)]
      {                                   // node:start[StartSimulation:in][ ... ]
        sc.set(0.0, 0.0, 1.0, 1.0);    // Set the color of subordinates to blue
        double r = sqrt(5000.0);                         // Range between tanks
        for (uint i=0; i<platoons.size(); ++i)        // Loop over the platoons
        {
          double x = 8500+r*((double) i);                             // Location
          sf.push_back(me);                                 // Create a new message
          sf[i].setPos(x,x);                            // Set the platoon position
          sf[i].setOri(-r, -r);        // Set the platoon orientation & spacing
          sf[i].setForm(LINE_ABREAST);                // Use line abreast formation
          if (i==0)
          {
            smove.setPos(x,x);                     // Move this platoon up slightly
            smove.setOri(-r, -r);
            smove.setForm(LINE_ABREAST);
          }

          mf.push_back(me);              // Create a new move formation message
          if (i<3)                        // If any of the first three platoons
          {
            x = -9500+r*((double) i);                      // Destination location
            mf[i].setPos(x,x);                          // Set the platoon position
            mf[i].setOri(-r, -r);      // Set the platoon orientation & spacing
            mf[i].setForm(LINE_ABREAST);              // Use line abreast formation
          }                                                            // if (i<3)
          else                                                        // if (i>=3)
          {
            double a = 0.125*PI;       // Angle to place tanks relative diagonal
            double s = sin(a);                        // Sin of angle from diagonal
            double h = 3500.0*sqrt(2.0+(s-2.0)*s);           // distance from CP
            double theta = 1.25*PI + (i==3 ? -a : a); // Angle for this platoon
            mf[i].setPos(9500+h*cos(theta), 9500+h*sin(theta));
            mf[i].setOri(-r, -r);      // Set the platoon orientation & spacing
            mf[i].setForm(V_FORMATION);               // Use line abreast formation
          }                                                // else from if (i<3)
        }                                     // for (uint i=0; i<sf.size(); ++i)
        slp.set(9500.0, 9500.0, 0.0);    // Set the position of the command post
      }                                   // node:start[StartSimulation:in][ ... ]
    }                                                                // mode:startup
  }                                               // process:BlueCompany(Company)
}
```

C.1.8. ChangeTrack.msg

```
{import message {TrackMotionEvent} }
{message:ChangeTrack(TrackMotionEvent);}
```

C.1.9. CommandPost.proc

```
{import process {SensorTrack} }
{import message {AddShape3D, SetColor, UnitSetup, RegisterEnvironmentObject} }
{import spt {sptEnvironmentObject} }

{
  process:CommandPost(SensorTrack)
  {                                         // process:CommandPost(SensorTrack)
    mode:Default
    {                                                                // mode:Default
      node:unitSetup[UnitSetup:in]                       // Setting the environment
                [RegisterEnvironmentObject:out=>(in.environment;),    // Reg
```

```
                    AddShape3D:as=>(in.getNode();),          // Ensure display
                    SetColor:sc=>(me;)]                        // Proper color
      {                 // node:unitSetup[UnitSetup:in][SetEnvironment:se]
        out.setForce(force = in.getForce());         // Set the force component
        out.setRadius(radius = in.getRadius());       // Set the sensor radius
        out.setMotion(nm);                 // Set the newtonian motion parameters
        as.add(me);                      // Add this as a subordinat to the node
        if (force==BLUE) sc.set(0.0, 0.0, 1.0);        // Set the proper color
        else if (force==RED) sc.set(1.0, 0.0, 0.0);
        else sc.set(1.0, 1.0, 1.0);
      }                 // node:unitSetup[UnitSetup:in][SetEnvironment:se]
    }                                                          // mode:Default
  }                                         // process:CommandPost(SensorTrack)
}
```

C.1.10. Company.proc

```
{import process {Platoon, CommandPost, Environment} }
{import message {SetEnvironment, UnitSetup, AddTrack, ChangeTrack, LoseTrack,
                 SetNewtonianMotion, RefreshDisplay, StartSimulation,
                 SetRefresh, Destroyed} }
{import spt {sptNewtonianMotion} }
{import gvm {gvmTacticalView, gvmTacticalGrid, gvmTrack, gvmAddTrack,
             gvmChangeTrack, gvmDeleteTrack, gvmRefresh} }
{import {"GLUTViewManager.h", <GL/glut.h>, <math.h>} }

{debug false}

{
  process:Company
  {                                                        // process:Company
    Platoon:platoons[5];               // The platoons composing this company
    bool:active[5](true);                     // All of the platoons are active
    CommandPost:cp;                            // Command post for the company
    process:environment;            // Environment in which the simulation exists
    ulong:force(NEUTRAL);                                         // Force flag
    double:tankSensorRange(2000.0);               // Default tank sensor range
    double:cpSensorRange(5000.0);         // Default command post sensor range
    ulong:trackCount[];                // Number of platoons tracking each object
    spt::NewtonianMotion:tracks[];               // Actual objects being tracked
    ulong:trackForce[];                       // Force association of the tracks
    gvm::TacticalView*:view;                     // Tactical view for the company
    double:refreshInterval(0.01);         // Time between consecutive refreshes

    method:init(public; void;)
    {                                           // method:init(public; void;)
      view = new gvm::TacticalView;                         // Create a new view
      dynamic_cast<sodl::GLUTViewManager&>(*EngineStand::stand.vm).
        addView(view);
    }                                           // method:init(public; void;)

    method:restore(public; void;)
    {                                        // method:restore(public; void;)
      view->restore(getTime());                 // Rollback the view to time t
    }                                        // method:restore(public; void;)

    method:fossilCollect(public; void;)
    {                                  // method:fossilCollect(public; void;)
      view->fossilCollect(getTime());   // Fossil collect the view up to time t
    }                                  // method:fossilCollect(public; void;)

    mode:startup
    {
      node:start[StartSimulation:in][RefreshDisplay:out=>(me;):(0.0)] {}

      node:unitSetup[SetEnvironment:in]              // Setting the environment
                   [UnitSetup:pl_us=>(platoons;),          // Setup the platoons
                    UnitSetup:cp_us=>(cp;)]           // Setup the command post
      {             // node:setEnvironment[SetEnvironment:in][SetEnvironment:se]
        environment=in.getEnvironment();            // Set subordinate environs
        pl_us.set(force, tankSensorRange, environment, in.getNode());
    cp_us.set(force, cpSensorRange, environment, in.getNode());
  }           // node:setEnvironment[SetEnvironment:in][SetEnvironment:se]

}

mode:run
{
  node:setNewtonianMotion[SetNewtonianMotion:in][]
  {                       // node:setNewtonianMotion[SetNewtonianMotion:in][]
    if (tracks.size() <= in.index)        // If we need to increase storage
    {
      tracks.resize(in.index+1);                   // Resize the track array
      trackCount.resize(in.index+1, 0);                // Resize the counter
      trackForce.resize(in.index+1, 0);                // Resize the counter
    }                                     // if (tracks.size() <= in.index)
    tracks[in.index] = in.nm;                   // Save the motion paramter

    if (trackCount[in.index]==0)           // Is this a new friendly track?
    {
      double rad=in.getSource().getType()==SPT_CommandPost ? cpSensorRange
                                                       : tankSensorRange;
      view->schedule(new gvm::AddTrack(*view, getTime(), in.index,
                     in.nm, rad, force));
      trackCount[in.index] = 1;         // Increment the number of trackers
      trackForce[in.index]=force;                     // Set track identity
    }                                       // if (trackCount[in.index]==0)
    else                                   // If we've seen this one before
    {
      view->schedule(new gvm::ChangeTrack(*view, getTime(), in.index,
                                          in.nm));
    }
  }                       // node:setNewtonianMotion[SetNewtonianMotion:in][]

  node:setRefreshInterval[SetRefresh:in][]
  { refreshInterval = in.refreshInterval; }

  node:refresh[RefreshDisplay:in]
              [RefreshDisplay:out=>(me;):(in.getTime()+refreshInterval)]
  {                  // node:refresh[RefreshDisplay:in][RefreshDisplay:out]
    view->schedule(new gvm::Refresh(*view,getTime()));  // Schedule refresh
  }                  // node:refresh[RefreshDisplay:in][RefreshDisplay:out]

  node:addTrack[AddTrack:in][]
  {                                         // node:addTrack[AddTrack:in][]
    if (tracks.size() <= in.getTrack())   // If we need to increase storage
    {
      tracks.resize(in.getTrack()+1);              // Resize the track array
      trackCount.resize(in.getTrack()+1, 0);           // Resize the counter
      trackForce.resize(in.getTrack()+1, 0);           // Resize the counter
    }                                // if (tracks.size() <= in.getTrack())

    if (trackCount[in.getTrack()]==0)
      view->schedule(new gvm::AddTrack(*view, getTime(), in.getTrack(),
                     in.motion, -1, in.getForce()));

    tracks[in.getTrack()] = in.motion;          // Save the motion paramter
    trackCount[in.getTrack()]++;        // Increment the number of trackers
    trackForce[in.getTrack()]=in.getForce();          // Set track identity
  }                                         // node:addTrack[AddTrack:in][]

  node:changeTrack[ChangeTrack:in][]
  {                                   // node:changeTrack[ChangeTrack:in][]
    if (tracks[in.getTrack()].getStartTime() != in.motion.getStartTime())
    {
      view->schedule(new gvm::ChangeTrack(*view,getTime(),in.getTrack(),
                     in.motion));
      tracks[in.getTrack()] = in.motion;        // Save the motion paramter
    }
  }                                   // node:changeTrack[ChangeTrack:in][]

  node:loseTrack[LoseTrack:in][]
```

```
      {                                     // node:loseTrack[LoseTrack:in][]
        --trackCount[in.getTrack()];      // Reduce the number of trackers by 1

        if (trackCount[in.getTrack()]==0)
          view->schedule(new gvm::DeleteTrack(*view,getTime(),
                                            in.getTrack()));
      }                                     // node:loseTrack[LoseTrack:in][]

      node:destroyed[Destroyed:in][]       // Upon notification of a unit's loss
      {                                     // node:destroyed[Destroyed:in][]
        if (in.getSource().getType() == SPT_CommandPost)     // If CP destroyed
        {
          std::cout << me << " lost." << std::endl;           // This team lost
          exit(0);                                         // Don't do this at home
        }                   // if (in.getSource().getType() == SPT_CommandPost)
        else                // The notification should have come from a platoon
        {
          if (in.index == ((ulong) -1))               // If the platoon was lost
          {
            bool a = false;           // Accumulator to determine company status
            for (ulong i=0; i<platoons.size(); ++i)   // Loop over all platoons
            {
              if (in.getSource()==platoons[i])        // If this platoon was it
                active[i]=false;                         // It is now inactive
              a = a || active[i];                       // Accumulate activity
            }                             // for (ulong i=0; i<platoons.size(); ++i)
            if (!a)                          // If all of the platoons are destroyed
            {
              std::cout << me << " surrenders." << std::endl; // This team lost
              exit(0);                                     // Don't do this at home
            }
          }                                     // if (in.index == ((ulong) -1))
          else                                  // if (in.index != ((ulong) -1))
          {
            trackCount[in.index]=((ulong) -1);        // No longer tracking tank
            view->schedule(new gvm::DeleteTrack(*view,getTime(),in.index));
          }
        }              // else from if (in.getSource.getType()==SPT_CommandPost)
      }                                     // node:destroyed[Destroyed:in][]
    }                                                          // mode:startup
  }                                                         // process:Company
}
```

C.1.11. Destroyed.msg

```
{import message {SetValue} }
{ message:Destroyed(SetValue); }
```

C.1.12. Environmnent.proc

```
{import message {RegisterEnvironmentObject, AddEnvironment, AddTrack,
                 LoseTrack, ChangeTrack, SetNewtonianMotion, ScheduleAddTrack,
                 ScheduleLoseTrack, Explosion, Impact, Hit, Destroyed} }
{import spt {sptLinearMotion, sptLinearMotion, sptEnvironmentObject, sptDefs} }
{import std {<map>} }
{import {<math.h>} }

{
  process:Environment
  {                                                     // process:Environment
    spt::EnvironmentObject:motion[];               // Object motion parameters
    process:objects[];                            // Associated process handles
    std::set<ulong>:tracks[];                // List of sensor/track associations
    bool:active[];                              // Active list of sensor/tracks

    method:trackTimes(public; std::vector<double>; spt::EnvironmentObject&:s;
                                                   spt::EnvironmentObject&:t;)
    {     // method:trackTimes(public; std::vector<double>; LinearMotion; ... )
      double time = getTime();                          // Get the current time
      spt::vertex p = t.lp(time)-s.lp(time);               // Relative position
      spt::vertex v = t.lv(time)-s.lv(time);               // Relative velocity
    std::vector<double> rv;                  // Return the detect/loss times
    double a = dot(v,v);                  // Get the polynomial coefficients
    double b = 2*dot(p,v);
    double c = dot(p,p)-s.rad()*s.rad();
    double disc = b*b-4*a*c;
    if (a > 0.0 && disc > 0.0)       // If there are more than one real roots
    {
      double t1 = time+(-b+sqrt(disc))/(2*a);
      double t2 = time+(-b-sqrt(disc))/(2*a);
      rv.push_back(min(t1, t2));             // Entry time into sensor range
      rv.push_back(max(t1, t2));              // Exit time from sensor range
      p = t.lp(t1)-s.lp(t1);
      p = t.lp(t2)-s.lp(t2);
    }                                          // if (a > 0.0, && disc > 0.0)
    else if (a==0.0 && c<0.0)    // If rel vel is 0 & track in range of sensor
    {           // Track has always been and will alway be in range of sensor range
      rv.push_back(-Clock::getEndTime());                    // A long time ago
      rv.push_back(2.0*Clock::getEndTime());                          // Forever
    }                                             // else if (a==0.0 && c<0.0)

    return rv;                                                // Return the times
  }     // method:trackTimes(public; std::vector<double>; LinearMotion; ... )

  method:configure(public; void; std::vector<AddTrack>&:at;
                                 ulong:sensor; ulong:track;)
  {       // method:configure(public; void; std::vector<AddTrack>&:at; ... )
    at.push_back(me);                                    // Generate new message
    at.back().addDest(objects[sensor]);         // The message goes to sensor
    at.back().setMotion(motion[track]);        // Set track motion parameters
    at.back().set(track, motion[track].iff());          // Set identifier info
  }       // method:configure(public; void; std::vector<AddTrack>&:at; ... )

  method:configure(public; void; std::vector<LoseTrack>&:lt;
                                 ulong:sensor; ulong:track;)
  {      // method:configure(public; void; std::vector<LoseTrack>&:at; ... )
    lt.push_back(me);                                    // Generate new message
    lt.back().addDest(objects[sensor]);         // The message goes to sensor
    lt.back().set(track, motion[track].iff());            // Set track id info
  }      // method:configure(public; void; std::vector<LoseTrack>&:at; ... )

  method:configure(public; void; std::vector<ScheduleAddTrack>&:at; double:t;
                                 ulong:sensor; ulong:track;)
  {  // method:configure(public; void; std::vector<ScheduleAddTrack>&:at;...)
    at.push_back(me);                                    // Generate new message
    at.back().addDest(me);                          // The message comes to me
    at.back().setTime(t);                                 // With time stamp t
    at.back().set(sensor, track);         // Set sensor and the track indices
  }  // method:configure(public; void; std::vector<ScheduleAddTrack>&:at;...)

  method:configure(public; void; std::vector<ScheduleLoseTrack>&:lt;
                                 double:t; ulong:sensor; ulong:track;)
  { // method:configure(public; void; std::vector<ScheduleLoseTrack>&:lt;...)
    lt.push_back(me);                                    // Generate new message
    lt.back().addDest(me);                          // The message comes to me
    lt.back().setTime(t);                                 // With time stamp t
    lt.back().set(sensor, track);     // Set the sensor and the track indices
  } // method:configure(public; void; std::vector<ScheduleLoseTrack>&:lt;...)

  method:testDetect(public; void; ulong:sensor; ulong:track;
                                  std::vector<ScheduleAddTrack>&:sat;
                                  std::vector<ScheduleLoseTrack>&:slt;
                                  std::vector<AddTrack>&:at;
                                  std::vector<LoseTrack>&:lt;)
  {                                     // method:testDetect(public; void; ...)
    std::vector<double> t = trackTimes(motion[sensor], motion[track]);
    double mt=min(motion[track].getStopTime(), motion[sensor].getStopTime());

    if (active[track] && active[sensor])
    {
      if (t.empty() &&                               // If sensor never sees track
          tracks[track].find(sensor) != tracks[track].end())       // If known
      {
        configure(lt, sensor, track);               // It is no longer known
      }                                            // if (t.empty() && ... )
      else if (!t.empty())                   // If there are detection times
      {
        if (t[0]<getTime() && t[1]>getTime())  // Is track currently in range
        {
          if (tracks[track].find(sensor)==tracks[track].end())  // Not known?
          {
            tracks[track].insert(sensor);      // Associate track with sensor
            configure(at, sensor, track);    // Configure sensor notification
          }           // if (tracks[track].find(sensor)==tracks[track].end())
          if (t[1]<mt)                      // Should we schedule a LoseTrack?
          {
            configure(slt,t[1],sensor,track); // Configure lose track message
          }                                                   // if (t[1]<mt)
        }                              // if (t[0]<getTime() && t[1]>getTime())
        else if (t[0]>getTime() && t[0]<mt)            // If entry will occur
        {
          configure(sat, t[0], sensor, track);     // Configure a new message
          if (t[1]<mt)                              // If the exit will occur
          {
            configure(slt, t[1], sensor, track);   // Configure a new message
          }                                                   // if (t[1]<mt)
        }                              // else if (t[0]>getTime() && t[0]<mt)
      }                                            // if (!oldTimes.empty())
    }                                // if (active[track] && active[sensor])
  }                                  // method:testDetect(public; void; ...)

  method:symetricDetect(public; void; ulong:s1; ulong:s2;
                                      std::vector<ScheduleAddTrack>&:sat;
                                      std::vector<ScheduleLoseTrack>&:slt;
                                      std::vector<AddTrack>&:at;
                                      std::vector<LoseTrack>&:lt;)
  {                                // method:symetricDetect(public; void; ...)
    std::vector<double> t = trackTimes(motion[s1], motion[s2]);
    double mt=min(motion[s1].getStopTime(), motion[s2].getStopTime());

    if (active[s1] && active[s2])
    {
      if (t.empty() &&                                 // If s1 never sees s2
          tracks[s2].find(s1) != tracks[s2].end())               // If known
      {
        configure(lt, s1, s2);                       // It is no longer known
        configure(lt, s2, s1);                       // It is no longer known
      }                                            // if (t.empty() && ... )
      else if (!t.empty())                   // If there are detection times
      {
        if (t[0]<getTime() && t[1]>getTime())  // Is track currently in range
        {
          if (tracks[s1].find(s2)==tracks[s1].end())         // and not known
          {
            tracks[s2].insert(s1);                    // Associate s2 with s1
            tracks[s1].insert(s2);                    // Associate s1 with s2
            configure(at, s1, s2);               // Configure s1 notification
            configure(at, s2, s1);               // Configure s2 notification
          }                     // if (tracks[s1].find(s2)==tracks[s1].end())
          if (t[1]<mt)                      // Should we schedule a LoseTrack?
          {
            configure(slt,t[1],s1,s2);        // Configure lose track message
            configure(slt,t[1],s2,s1);        // Configure lose track message
          }                                                   // if (t[1]<mt)
        }                              // if (t[0]<getTime() && t[1]>getTime())
        else if (t[0]>getTime() && t[0]<mt)            // If entry will occur
        {
          configure(sat, t[0], s1, s2);            // Configure a new message
          configure(sat, t[0], s2, s1);            // Configure a new message
         if (t[1]<mt)                               // If the exit will occur
          {
            configure(slt, t[1], s1, s2);          // Configure a new message
            configure(slt, t[1], s2, s1);          // Configure a new message
      }                                               // if (t[1]<mt)
     }                               // else if (t[0]>getTime() && t[0]<mt)
    }                                            // if (!oldTimes.empty())
  }                                        // if (active[s1] && active[2])
}                               // method:symetricDetect(public; void; ...)

method:canSense(public; bool; ulong:i;)          // Index of sensing object
{                                // method:canSense(public; bool; ulong:i;)
  return (motion[i].iff() == RED || motion[i].iff() == BLUE) && active[i];
}                                // method:canSense(public; bool; ulong:i;)

method:test(public; void; std::string:s;)
{
}

method:genTrackMessages(public; void; ulong:changed;  // Changed obj index
                                std::vector<ScheduleAddTrack>&:sat;
                                std::vector<ScheduleLoseTrack>&:slt;
                                std::vector<AddTrack>&:at;
                                std::vector<LoseTrack>&:lt;)
{            // method::genTrackMessages(public; void; ulong:changed; ... )
  ulong force = motion[changed].iff();                   // Get the force
  if (canSense(changed))              // If changed object can have a sensor
  {
    for (ulong i=0; i<motion.size(); ++i)    // Loop over the other objects
    {
      double iforce = motion[i].iff();          // Get next object's force
      if ( canSense(i) && force!=iforce )              // Is this sensible
      {                               // If forces are opposed to each other
        if (motion[i].rad() == motion[changed].rad())   // Do both at once?
          symetricDetect(i, changed, sat, slt, at, lt);
        else                  // If the sensor radii differ between objects
        {
          testDetect(i,changed,sat,slt,at,lt);     // Check with i as sensor
          testDetect(changed,i,sat,slt,at,lt); // Check with changed sensor
        }       // else from if (motion[i].rad() == motion[changed].rad())
      }                               // if (canSense(i) && force!=iforce)
      else if (iforce != force)   // If motion[i] has no sensing capability
      {
        testDetect(changed,i,sat,slt,at,lt); // Check with motion[changed]
      }
    }                              // for (ulong i=0; i<motion.size(); ++i)
  }                                             // if (canSense(changed))
  else if (active[changed])           // If changed is not a sensing object
    for (ulong i=0; i<motion.size(); ++i)      // Loop over everything else
      if (canSense(i))                                  // Is it a sensor?
        testDetect(i,changed,sat,slt,at,lt);  // Test with changed as track
}           // method::genTrackMessages(public; void; ulong:changed; ... )

method:inRange(public; bool; ulong:sensor; ulong:track;)
{             // method:inRange(public; bool; ulong:sensor; ulong:track;)
  double adjTime = getTime() + 0.0001;          // Get time slightly ahead
  double d = motion[sensor].rad()*motion[sensor].rad();  // Square of range
  spt::vertex rp = motion[track].lp(adjTime)- motion[sensor].lp(adjTime);
  return (dot(rp, rp) <= d);                      // Is it really in range?
}             // method:inRange(public; bool; ulong:sensor; ulong:track;)

mode:Default
{                                                         // mode:Default
  node:registerObject[RegisterEnvironmentObject:in]         // Request both
                   [AddEnvironment:out=>(in.getSource();),      // Confirm
                    ScheduleAddTrack:sat[],          // Pending detections
                    ScheduleLoseTrack:slt[],       // Pending track losses
                    AddTrack:at[],              // Add track notifications
                    LoseTrack:lt[] ]           // Lose track notifications
  {       // node:registerObject[RegisterEnvironmentObject][AddEnvironment]
    out.index = motion.size();          // Report back to source its index
    motion.push_back(spt::EnvironmentObject(in.getForce(),
                                        in.getRadius()));     // new s_t
    static_cast<spt::NewtonianMotion&>(motion.back())=in.getMotion();
    objects.push_back(in.getSource());           // Save the process handle
  tracks.push_back(std::set<ulong>());              // Create a new one
  tracks.back().clear();                                // Clear the set
  active.push_back(true);                   // Track is currently active
  genTrackMessages(motion.size()-1, sat, slt, at, lt); // Update messages
}       // node:registerObject[RegisterEnvironmentObject][AddEnvironment]

node:setNewtonianMotion[SetNewtonianMotion:in]     // Something is moving
                        [ScheduleAddTrack:sat[],      // Schedule detects
                         ScheduleLoseTrack:slt[],      // Scheduled loses
                         AddTrack:at[],                // Current detects
                         LoseTrack:lt[],               // Current detects
                         ChangeTrack:ct]     // Change the track direction
{               // node:setNewtonianMotion[SetNewtonianMotion:in][ ... ]
  static_cast<spt::NewtonianMotion&>(motion[in.index])=in.get();
  genTrackMessages(in.index,sat,slt,at,lt);              // Update things
  std::set<ulong>::iterator i;   // Index for sensors knowing about track
  for (i=tracks[in.index].begin(); i!=tracks[in.index].end(); ++i)
    ct.addDest(objects[*i]);     // Notify the sensors tracking the track
  ct.setMotion(motion[in.index]);                // Set motion parameters
  ct.set(in.index,motion[in.index].iff());         // Set track ID params
}               // node:setNewtonianMotion[SetNewtonianMotion:in][ ... ]

node:addTrack[ScheduleAddTrack:in]
             [AddTrack:out=>(objects[in.getSensor()];)]
{            // node:addTrack[AddTrack:in][AddTrack:out=>(in.getSensor()]
  ulong track = in.getTrack();                     // Get the track index
  ulong sensor = in.getSensor();                  // Get the sensor index
  bool ir = inRange(sensor, track) && active[sensor] && active[track];
  bool im = tracks[track].find(sensor) == tracks[track].end();
  if (ir && im)              // If in range & not alreay known to sensor
  {
    out.setMotion(motion[track]);           // Set track motion paramters
    out.set(track, motion[track].iff());           // Set identifier info
    tracks[track].insert(sensor);                  // Register the sensor
  }                               // if (inRange(sensor, track) && ... )
  else out.setTX(false);                          // Don't send the message
}            // node:addTrack[AddTrack:in][AddTrack:out=>(in.getSensor()]

node:loseTrack[ScheduleLoseTrack:in]
              [LoseTrack:out=>(objects[in.getSensor()];)]
{         // node:loseTrack[LoseTrack:in][LoseTrack:out=>(in.getSensor()]
  ulong track = in.getTrack();                     // Get the track index
  ulong sensor = in.getSensor();                  // Get the sensor index

  bool ir = inRange(sensor, track) && active[sensor] && active[track];
  bool im = tracks[track].find(sensor) == tracks[track].end();
  if (!ir && !im)                      // If out of range & known to sensor
  {
    out.set(track, motion[track].iff());        // Format message payload
    tracks[track].erase(sensor);       // Remove sensor from tracker list
  }                               // if (inRange(sensor, track) && ... )
  else out.setTX(false);                          // Don't send the message
}         // node:loseTrack[LoseTrack:in][LoseTrack:out=>(in.getSensor()]

node:explosion[Explosion:in]
              [Impact:out[],Hit:h=>(in.getSource();)]
{                    // node:explosion[Explosion:in][Impact:out[], Hit:h]
  for (ulong i=0; i<motion.size(); ++i)    // Loop over all of the objects
  {
    spt::vertex diff = in.get()-motion[i].lp(getTime());
    if (dot(diff, diff)<9.0)          // If they were within 3m of impact
    {
      out.push_back(me);   // Add message to inform the target it was hit
      out.back().addDest(objects[i]);       // Add object as a destination
      h.add(i);                       // Set the track index of object hit
    }                                      // if (dot(diff, diff)<9.0)
  }                            // for (ulong i=0; i<motion.size(); ++i)
}                    // node:explosion[Explosion:in][Impact:out[], Hit:h]

node:destroyed[Destroyed:in]                   // Something was destroyed
              [LoseTrack:out]                  // A bunch of track losses
```

```
        {                        // node:destroyed[Destroyed:in][LoseTrack:out[]]
          if (active[in.index])            // If the object is already inactive
          {
            active[in.index] = false;               // Object can no longer sense
            out.set(in.index, motion[in.index].iff());     // Set the track index
            std::set<ulong>::iterator i;    // Used to loop over sensors of track
            for (i=tracks[in.index].begin(); i!=tracks[in.index].end(); ++i)
              out.addDest(objects[*i]);                     // Inform the sensors
            tracks[in.index].clear();                // Delete the set of tracks
          }
          else out.setTX(false);                 // Don't transmit output message
        }                        // node:destroyed[Destroyed:in][LoseTrack:out[]]
      }                                                         // mode:Default
    }                                                    // process:Environment
  }
```

C.1.13. Explosion.msg

```
{import message {SetValue} }
{import spt {sptDefs} }

{
  message:Explosion(SetValue)
  {                                                      // message:Explosion
    spt::vertex:pos(0.0, 3);               // Initial position of the munition

    method:set(public; void; spt::vertex:p;) { pos=p; }
    method:get(public; spt::vertex;) { return pos; }
  }                                                      // message:Explosion
}
```

C.1.14. Fire.msg

```
{import spt {sptDefs} }

{
  message:Fire
  {                                                           // message:Fire
    double:muzzle_velocity;                 // Muzzle velocity of the munition
    double:azimuth;                // Azimuth of the projectile motion (in radians)
    double:elevation;            // Elevation of the projectile motion (in radians)
    spt::vertex:pos(0.0, 3);       // Initial position of the munition
    spt::vertex:vel(0.0, 3);     // Velocity vector of firing platform

    method:set(public; void; double:mv;                      // Muzzle velocity
                             double:a;                               // Azimuth
                             double:e;                             // Elevation
                             spt::vertex:p;                         // Position
                             spt::vertex:v;)                        // Velocity
    {           // method:set(public; void; double; double; double; double[];)
      muzzle_velocity = mv;                         // Save the muzzle velocity
      azimuth = a;                                          // Save the azimuth
      elevation = e;                                      // Save the elevation
      pos = p;                                             // Save the position
      vel = v;                                             // Save the velocity
    }           // method:set(public; void; double; double; double; double[];)
  }                                                           // message:Fire
}
```

C.1.15. FormationMove.msg

```
{message:FormationMove;}
```

C.1.16. Ground.proc

```
{import process {NewtonianMotion} }
{process:Ground(NewtonianMotion);}
```

C.1.17. Hit.msg

```
{import std {<vector>} }

{
  message:Hit
  {                                                                // message:Hit
    ulong:track[];              // Track number of object, -1 if nothing was hit
    method:add(public; void; ulong:t;) { track.push_back(t); } // Add the track
    method:get(public; std::vector<ulong>;) { return track; }  // Report tracks
  }                                                                // message:Hit
}
```

C.1.18. HoldPosition.msg

```
{message:HoldPosition;}
```

C.1.19. Impact.msg

```
{message:Impact;}
```

C.1.20. LoseTrack.msg

```
{import message {TrackEvent} }
{message:LoseTrack(TrackEvent);}
```

C.1.21. MoveFormation.msg

```
{import message {AdjustFormation} }
{message:MoveFormation(AdjustFormation);}
```

C.1.22. MoveTo.msg

```
{import spt {sptDefs} }

{
  message:MoveTo
  {                                                             // message:MoveTo
    spt::vertex:pos(0.0, 3);
    spt::vertex:ori(0.0, 3);

    method:fixOri(protected; spt::vertex; spt::vertex:o;)
    {                  // method:fixOri(protected; spt::vertex; spt::vertex:o;)
      for (ulong i=0; i<3; ++i)                            // Loop over the axes
      {
        o[i] = fmod(o[i], PI*2.0);                         // Adjust angular pos
        if (o[i]<0.0) o[i]+=PI*2.0;                      // Make certain it's > 0
      }                                                    // for (i=0; i<3; ++i)
      return o;                         // Return the adjusted orientation array
    }                  // method:fixOri(protected; spt::vertex; spt::vertex:o;)

    method:setPos(public; void; spt::vertex:v;) { pos = v; }
    method:setPos(public; void; double:x; double:y; double:z;)
    { pos[0]=x; pos[1]=y; pos[2]=z;  }
    method:setPos(public; void; double:x; double:y;)
    { pos[0]=x; pos[1]=y; pos[2]=0.0;  }

    method:setOri(public; void; spt::vertex:v;) { ori = fixOri(v); }
    method:setOri(public; void; double:x; double:y; double:z;)
    { ori[0]=x; ori[1]=y; ori[2]=z; ori=fixOri(ori); }

    method:set(public; void; spt::vertex:p; spt::vertex:o;)
    { pos=p; ori=fixOri(o); }

    method:getPos(public; spt::vertex;) { return pos; }
    method:getOri(public; spt::vertex;) { return ori; }
  }                                                             // message:MoveTo
}
```

C.1.23. MovementComplete.msg

```
{message:MovementComplete;}
```

C.1.24. Munition.proc

```
{import process {NewtonianMotion} }
{import message {Fire, Impact, SetNewtonianMotion, Explosion,
                 StartSimulation, SetEnvironment, Hit, AddShape3D} }
{import {<math.h>} }

{
  process:Munition(NewtonianMotion)
  {                                          // process:Munition(NewtonianMotion)
    process:environment;            // Environment in which the munition exists
    process:parent;                                           // Parent process
    process:grNode;                       // Graphics node for display purposes

    method:getImpactTime(public; double; double:v; double:a;)
    {             // method:getImpactTime(public; double; double:v; double:a;)
      return -2.0*v/a;                   // Return the value to the calling routine
    }             // method:getImpactTime(public; double; double:v; double:a;)

    mode:Default
    {                                                             // mode:Default
      node:start[StartSimulation:in][]
      {                                        // node:start[StartSimulation:in][]
        fired.setActive(false);                        // Turn the "fired" mode off
      }                                        // node:start[StartSimulation:in][]
    }                                                             // mode:Default

    mode:waiting
    {                                                             // mode:waiting
      node:setEnvironment[SetEnvironment:in][]
      {                               // node:setEnvironment[SetEnvironment:in][]
        environment = in.getEnvironment();  // Environment in which this exists
        grNode = in.getNode();          // Graphics node displaying the munition
        parent = in.getSource();              // Save the parent process handle
      }                               // node:setEnvironment[SetEnvironment:in][]

      node:fire[Fire:f]                         // Request to fire the munition
              [Impact:imp=>(me;),                         // Schedule the impact
               AddShape3D:as=>(grNode;),       // Start rendering the munition
               SetNewtonianMotion:out[]}                  // Set motion parameters
      {             // node:fire[Fire:f][Impact,AddShape3DSetNewtonianMotion]
        spt::vertex v(0.0, 3), a(0.0, 3), p(f.pos);
        v[0] = f.muzzle_velocity*cos(f.elevation)*cos(f.azimuth)+f.vel[0];
        v[1] = f.muzzle_velocity*cos(f.elevation)*sin(f.azimuth)+f.vel[1];
        v[2] = f.muzzle_velocity*sin(f.elevation)+f.vel[2];
        a[0] = a[1] = 0.0;
        a[2] = -0.9;                               // Acceleration due to gravity

        imp.setTime(getTime()+getImpactTime(v[2],-9.8));          // Impact time
        nm.setLM(p, v, a, getTime(), imp.getTime());   // Set motion parameters

        notify(out);                    // Inform interested parties about this
        waiting.setActive(false);                        // We're no longer waiting
        fired.setActive(true);                                 // We are now firing

        as.add(me);  // Add this process as a subordinate to the rendering node
      }              // node:fire[Fire:f][Impact,AddShape3DSetNewtonianMotion]
    }                                                             // mode:waiting

    mode:fired
    {                                                               // mode:fired
      node:impact[Impact:in]                     // We impacted the environment
                [Explosion:explode[],                    // An explosion occurred
                 SetNewtonianMotion:out[]]                          // Stop motion
      {                            // node:impact[Impact:in][Explosion:explode]
        double t = getTime();                        // Get the time of the impact
        spt::vertex p(nm.lp(t));                            // Get current position
        nm.lv(0.0, 0.0, 0.0, t);            // Set to stop at current position
        nm.la(0.0, 0.0, 0.0, t);            // Set to stop at current position
        notify(out);                     // Inform interested parties about this

        std::map<process, gvm::object_index>::iterator i;     // For loop index
        for (i=views.begin(); i!=views.end(); ++i)       // Loop over index map
        {
          explode.push_back(me);                         // Allocate a new message
          explode.back().addDest(i->first);   // Add this view as a destination
          explode.back().index = i->second;                 // Specify the index
          explode.back().set(p);                         // Specify the position
        }                           // for (i=views.begin(); i!=views.end(); ++i)

        explode.push_back(me);                           // Allocate a new message
        explode.back().addDest(environment);               // Add environ as dest
        explode.back().set(p);                           // Specify the position
      }                            // node:impact[Impact:in][Explosion:explode]

      node:hit[Hit:in][Hit:out=>(parent;)]         // Notify parent what we hit
      {                                 // node:hit[Hit:in][Hit:out=>(parent;)]
        out.track = in.track;                             // Set the target data
        fired.setActive(false);                            // No more adventures
      }                                 // node:hit[Hit:in][Hit:out=>(parent;)]
    }                                                             // mode:fired
  }                                      // process:Munition(NewtonianMotion)
}
```

C.1.25. NewtonianMotion.proc

```
{import process {Shape3D} }
{import message {SetNewtonianMotion, SetLinearPosition, SetLinearVelocity,
                 SetLinearAcceleration, SetAngularPosition, SetAngularVelocity,
                 SetAngularAcceleration, AddView} }
{import {<math.h>} }
{import std {<valarray>} }
{import spt {sptNewtonianMotion, sptLinearMotion, sptAngularMotion} }

{
  process:NewtonianMotion(Shape3D)
  {                                        // process:NewtonianMotion(Shape3D)
    spt::NewtonianMotion:nm;                  // Parameters of Newtonian Motion

    method:notify(public; void; std::vector<SetNewtonianMotion>&:out;)
    {     // method:notify(public; void; std::vector<SetNewtonianMotion>&:out;)
      std::map<process, gvm::object_index>::iterator i;       // For loop index
      for (i=views.begin(); i!=views.end(); ++i)        // Loop over index map
      {
        out.push_back(me);                           // Allocate a new message
        out.back().addDest(i->first);          // Add this view as a destination
        out.back().index = i->second;                       // Specify the index
        out.back().set(nm);                // Specify the Linear Motion paramters
      }                         // for (i=views.begin(); i!=views.end(); ++i)
    }     // method:notify(public; void; std::vector<SetNewtonianMotion>&:out;)

    mode:Default
    {                                                           // mode:Default
      node:addView[AddView:in]
                  [SetNewtonianMotion:snm=>(in.getSource();)]
      {                                        // Node:addView[AddView:in][...]
        snm.set(nm);                              // Set the motion parameters
        snm.index = in.index;                                   // Get the index
      }                                        // Node:addView[AddView:in][...]

      node:setLinearPosition[SetLinearPosition:in]
                            [SetNewtonianMotion:out[]]
      {                // node:setLinearPosition[SetLinearPosition:in][ ... ]
        nm.lp(in.get(), getTime());         // Copy the linear position contents
        notify(out);                                // Notify the views, et al
      }                // node:setLinearPosition[SetLinearPosition:in][ ... ]

      node:setLinearVelocity[SetLinearVelocity:in]
```

```
                          [SetNewtonianMotion:out[]]
      {               // node:setLinearVelocity[SetLinearVelocity:in][ ... ]
        nm.lv(in.get(), getTime());      // Copy the linear velocity contents
        notify(out);                              // Notify the views, et al
      }               // node:setLinearVelocity[SetLinearVelocity:in][ ... ]

      node:setLinearAcceleration[SetLinearAcceleration:in]
                              [SetNewtonianMotion:out[]]
      {          // node:setLinearAcceleration[SetLinearAcceleration:in][ ... ]
        nm.la(in.get(), getTime());    // Copy the linear acceleration contents
        notify(out);                              // Notify the views, et al
      }          // node:setLinearAcceleration[SetLinearAcceleration:in][ ... ]

      node:setAngularPosition[SetAngularPosition:in]
                            [SetNewtonianMotion:out[]]
      {             // node:setAngularPosition[SetAngularPosition:in][ ... ]
        nm.ap(in.get(), getTime());      // Copy the angular position contents
        notify(out);                              // Notify the views, et al
      }             // node:setAngularPosition[SetAngularPosition:in][ ... ]

      node:setAngularVelocity[SetAngularVelocity:in]
                            [SetNewtonianMotion:out[]]
      {             // node:setAngularVelocity[SetAngularVelocity:in][ ... ]
        nm.av(in.get(), getTime());      // Copy the angular velocity contents
        notify(out);                              // Notify the views, et al
      }             // node:setAngularVelocity[SetAngularVelocity:in][ ... ]

      node:setAngularAcceleration[SetAngularAcceleration:in]
                               [SetNewtonianMotion:out[]]
      {       // node:setAngularAcceleration[SetAngularAcceleration:in][ ... ]
        nm.aa(in.get(), getTime());   // Copy the angular acceleration contents
        notify(out);                              // Notify the views, et al
      }       // node:setAngularAcceleration[SetAngularAcceleration:in][ ... ]

      node:setNewtonianMotion[SetNewtonianMotion:in]
                            [SetNewtonianMotion:out[]]
      {             // node:setNewtonianMotion[SetNewtonianMotion:in][ ... ]
        nm = in.get();                 // Copy the newtonian motion parameters
        notify(out);                       // Generate notification messages
      }       // node:setAngularAcceleration[SetAngularAcceleration:in][ ... ]
    }                                                       // mode:Default
  }                                     // process:NewtonianMotion(Shape3D)
}
```

C.1.26. Platoon.proc

```
{import process {Tank} }
{import message {AddShape3D, SetFormation, StartSimulation, Destroyed,
                 SetLinearPosition, SetAngularPosition, UnitSetup,
                 MoveFormation, MovementComplete, MoveTo, UnitSetup,
                 AddTrack, ChangeTrack, LoseTrack, SetNewtonianMotion} }
{import spt {sptEnvironmentObject} }
{import std {<valarray>} }
{import {<math.h>} }

{
  process:Platoon
  {                                                    // process:Platoon
    Tank:tanks[5];                             // The tanks in the platoon
    bool:active[5](true);              // Which of the tanks are still alive
    bool:complete[5](true);            // Phase complete flags for each tank
    process:environment;          // Environment in which the simulation occurs
    process:parent;                                // Parent unit (company)
    spt::vertex:pos(3);                                // Current position
    spt::vertex:ori(3);                             // Current orientation
    spt::vertex:destPos(3);                         // Destination position
    spt::vertex:destOri(3);                      // Destination orientation
    double:destLeft;                       // Destination line of left flank
    double:destRight;                     // Destination line of right flank
    double:startTime(-1.0);                         // Time we start moving
    double:left(0.0);                                    // Left flank line
```

```
double:right(0.0);                                         // Right flank line
ulong:trackCount[];                            // Number of members tracking track
spt::NewtonianMotion:tracks[];                          // List of all known tracks
ulong:force(UNKNOWN);                  // Which force is this platoon a member of

method:init(public; void;)
{                                                 // method:init(public; void;)
  turn_to_dest.setActive(false);                  // Turning to the destination
  move_to_dest.setActive(false);            // We're not moving to destination yet
  turn_to_heading.setActive(false);            // Not turning to final heading yet
}                                                 // method:init(public; void;)

method:phaseDone(public; bool; process:p;)
{                             // method:phaseDone(public; bool; process:p;)
  bool done=true;                                          // Are we done yet?
  for (uint i=0; i<tanks.size(); ++i)                    // Loop over the tanks
  {
    if (p==tanks[i]) complete[i]=true;                     // Is this the one?
    done &= (complete[i] || !active[i]);                    // Are we done yet
  }                                     // for (uint i=0; i<tanks.size(); ++i)
  return done;                    // Return the value to the calling routine
}                             // method:phaseDone(public; bool; process:p;)

method:getOrientation(protected; spt::vertex; spt::vertex:dO;)  // Dest ori
{                                                     // method:getOrientation
  double theta = atan2(dO[1], dO[0]);                          // Get the angle
  spt::vertex rv(0.0, 3);                              // Resulting orientation

  rv[2] = theta;                                           // Direction to look
  return rv;                                          // Return the orientation
}                                                     // method:getOrientation

method:getPosition(protected; spt::vertex;
                   spt::vertex:dP;                      // Destination position
                   spt::vertex:dO;                   // Destination orientation
                   double:l;                              // Formation left leg
                   double:r;                             // Formation right leg
                   ulong:i;)                                             // Index
{                                                        // method:getPosition
  double c = ((double) i)-((double) (tanks.size()-1))/2.0; // Rel. location
  double h = norm(dO);                            // Get orientation magnitude
  double theta = atan2(dO[1], dO[0]);                          // Get the angle
  double act=0;                                          // Abreast factor in x
  double ast=0;                                          // Abreast factor in y
  spt::vertex rv(0.0, 3);                                  // Resize positions

  double leg = 0.5*PI+(c>0 ? -l : r);                              // Direction
  act=(c==0 ? 0.0 : h*cos(theta+leg));                      // V leg factor in x
  ast=(c==0 ? 0.0 : h*sin(theta+leg));                      // V leg factor in y

  rv[0]=dP[0]+c*act;                                        // Set the position
  rv[1]=dP[1]+c*ast;

  return rv;                  // Return the position to the calling routine
}                                                        // method:getPosition

mode:startup
{                                                               // mode:startup
  node:unitSetup[UnitSetup:in]                      // Setting the environment
                [UnitSetup:se=>(tanks;),               // Establish environment
                 AddShape3D:out=>(in.getNode();)]         // Add tanks to node
  {             // node:setEnvironment[UnitSetup:in][SetEnvironment:se...]
    environment = se.environment = in.environment;   // Set the environment
    force = in.getForce();                                  // This is our force
    parent = in.getSource();                          // Set the parent process
    se.set(force,in.getRadius(),environment,in.getNode());    // Setup tanks
    for (uint i=0; i<tanks.size(); ++i)           // Loop over all of the tanks
      out.add(tanks[i]);                           // Add each one to the scene
    startup.setActive(false);     //
  }             // node:setEnvironment[UnitSetup:in][SetEnvironment:se...]
}

mode:run
{
  node:setFormation[SetFormation:in]                    // Orders from above
                  [SetLinearPosition:slp[5]=>(tanks[@];),  // Set tank pos
                   SetAngularPosition:sap[5]=>(tanks[@];)] // Orient tanks
  {                            // node:setFormation[SetFormation:in][ ... ]
    left = in.getLeft();                        // Get formation left flank
    right = in.getRight();                     // Get formation right flank
    pos = in.getPos();                   // This is now the current position
    ori = in.getOri();                // This is now the current orientation

    for (uint i=0; i<tanks.size(); ++i)              // Loop over the tanks
    {
      slp[i].set(getPosition(pos, ori, left, right, i));    // Get position
      sap[i].set(getOrientation(ori));              // Get the orientation
    }                                 // for (uint i=0; i<tanks.size(); ++i)
  }                            // node:setFormation[SetFormation:in][ ... ]

  node:moveFormation[MoveFormation:in]      // Request to move the formation
                   [MoveTo:out[5]=>(tanks[@];)]           // Move the tanks
  {                  // node:moveFormation[MoveFormation:in][MoveTo:out[5]]
    destLeft = in.getLeft();             // Destination formation left flank
    destRight = in.getRight();           // Destination formation left flank
    destOri = in.getOri();                        // Destination orientation
    destPos = in.getPos();                           // Destination position
    if (startTime>0) pos += 40.0*(getTime()-startTime)*ori;    // If moving
    spt::vertex rp = destPos-pos;                       // Relative position

    startTime = -1;                            // We are not moving any more
    double theta = atan2(rp[1], rp[0]);  // Relative bearing to destination
    double dist = norm(ori);             // Distance between adjacent units

    ori[0] = dist*cos(theta);                       // Modify the orientation
    ori[1] = dist*sin(theta);
    ori[2] = 0.0;

    for (uint i=0; i<tanks.size(); ++i)              // Loop over the tanks
    {
      out[i].setPos(getPosition(pos, ori, left, right, i));
      out[i].setOri(getOrientation(ori));
    }                                 // for (uint i=0; i<tanks.size(); ++i)

    turn_to_dest.setActive(true);              // Turning to the destination
    move_to_dest.setActive(false);   // We're not moving to destination yet
    turn_to_heading.setActive(false);   // Not turning to final heading yet

    for (uint i=0; i<complete.size(); ++i) complete[i]=false;
  }                  // node:moveFormation[MoveFormation:in][MoveTo:out[5]]

  node:setNewtonianMotion[SetNewtonianMotion:in]
                         [SetNewtonianMotion:out=>(parent;)]
  {      // node:setNewtonianMotion]SetNewtonianMotion][SetNewtonianMotion]
    out.set(in.nm); // Set the newtonian motion paramters of output message
    out.index = in.index;                       // Set the index value, too

    if (tracks.size() <= in.index)        // If we need to increase storage
    {
      tracks.resize(in.index+1);                    // Resize the track array
      trackCount.resize(in.index+1, 0);                 // Resize the counter
    }                                     // if (tracks.size() <= in.index)
    tracks[in.index] = in.nm;                    // Save the motion paramter
  }      // node:setNewtonianMotion]SetNewtonianMotion][SetNewtonianMotion]

  node:addTrack[AddTrack:in][AddTrack:out=>(parent;)]
  {                              // node:addTrack[AddTrack:in][AddTrack:out]
    if (tracks.size() <= in.getTrack())   // If we need to increase storage
    {
      tracks.resize(in.getTrack()+1);               // Resize the track array
      trackCount.resize(in.getTrack()+1, 0);            // Resize the counter
    }                                 // if (tracks.size() <= in.getTrack())

    if (trackCount[in.getTrack()]++>0)         // If this is not a new track
      out.setTX(false);                     // Don't bother informing parent
    else                                              // If the track is new
    {
      tracks[in.getTrack()] = in.getMotion();   // Save the motion paramter
      out.setMotion(in.getMotion());          // Notify as to object motion
      out.set(in.getTrack(), in.getForce());      // Provide force tracking
    }                          // else from if (trackCount[in.getTrack()]>1)
  }                              // node:addTrack[AddTrack:in][AddTrack:out]

  node:changeTrack[ChangeTrack:in][ChangeTrack:out=>(parent;)]
  {                     // node:changeTrack[ChangeTrack:in][ChangeTrack:out]
    ulong t = in.getTrack();                          // Get the track index

    if (in.getMotion().getStartTime()==tracks[t].getStartTime())  // Known?
      out.setTX(false);                      // We already informed the parent
    else                                          // If this is a new update
    {
      tracks[t] = in.getMotion();               // Save the motion paramter
      out.setMotion(in.getMotion());          // Notify as to object motion
      out.set(t,in.getForce());                   // Provide force tracking
    }                 // else from if (in.getMotion().getStartTime == ... )
  }                     // node:changeTrack[ChangeTrack:in][ChangeTrack:out]

  node:loseTrack[LoseTrack:in][LoseTrack:out=>(parent;)]
  {                           // node:loseTrack[LoseTrack:in][LoseTrack:out]
    if (--trackCount[in.getTrack()]>0)     // If there are tracks remaining
      out.setTX(false);            // Don't notify parent that track is lost
    else                     // If this was the last one observing the track
      out.set(in.getTrack(),in.getForce());           // Provide force info
  }                           // node:loseTrack[LoseTrack:in][LoseTrack:out]

  node:destroyed[Destroyed:in]
                [Destroyed:out[]=>(parent;)]
  {                         // node:destroyed[Destroyed:in][Destroyed:out[]]
    bool a = false;              // Accumulator for testing state of platoon

    out.push_back(me);                // Inform the company of the tank loss
    out.back().index = in.index;           // Tell company which one it was

    for (ulong i=0; i<tanks.size(); ++i)       // Loop over tanks in platoon
    {
      if (tanks[i]==in.getSource())   // If the ith tank just got destroyed
      {
        active[i]=false;           // This is no longer part of the platoon
      }
      a = a || active[i];          // Accumulate to see if we're still alive
    }                                        // for (i=0; i<tanks.size(); ++i)

    if (!a)                                    // If the platoon is destroyed
    {
      out.push_back(me);              // Inform company if platoon destroyed
      run.setActive(false);              // This platoon is no longer active
      turn_to_dest.setActive(false);           // Turning to the destination
      move_to_dest.setActive(false); // We're not moving to destination yet
      turn_to_heading.setActive(false); // Not turning to final heading yet
    }
  }                         // node:destroyed[Destroyed:in][Destroyed:out[]]
}                                                          // mode:startup

mode:turn_to_dest
{                                                    // mode:turn_to_dest
  node:movementComplete[MovementComplete:in]                 // One is done
                       [MoveTo:out[5]=>(tanks[@];)]  // Next movement phase
  {           // node:movementComplete[MovementComplete:in][MoveTo[5]:out]
    if (phaseDone(in.getSource()))         // If we can go to the next phase
    {
      for (uint i=0; i<out.size(); ++i)     // Loop over the output messages
      {
          out[i].setPos(getPosition(destPos, ori, destLeft, destRight, i));
          out[i].setOri(getOrientation(ori));
          complete[i]=false;                        // Not done with next phase
        }                                   // for (int i=0; i<out.size(); ++i)
        turn_to_dest.setActive(false);      // No longer turning to destination
        move_to_dest.setActive(true);          // Start moving to destination
        turn_to_heading.setActive(false);     // Not turning to final heading
        startTime = getTime();                         // Get the current time
      }                                     // if (phaseDone(in.getSource())
      else                  // if we're not yet done turing to final heading
        for (uint i=0; i<out.size(); ++i) out[i].setTX(false);  // Don't send
    }   // node:movementComplete[MovementComplete:in][MoveTo[5]=>(tanks[@];)]
  }                                                       // mode:turn_to_dest

  mode:move_to_dest
  {                                                       // mode:move_to_dest
    node:movementComplete[MovementComplete:in]                   // One is done
                          [MoveTo:out[5]=>(tanks[@];)]  // Next movement phase
    {            // node:movementComplete[MovementComplete:in][MoveTo[5]:out]
      if (phaseDone(in.getSource()))        // If we can go to the next phase
      {
        pos = destPos;                                // We're at our final dest
        startTime = -1;                                  // Not moving any more
        left = destLeft;                  // We are now in the final formation
        right = destRight;                // We are now in the final formation
        for (uint i=0; i<out.size(); ++i)    // Loop over the output messages
        {
          out[i].setPos(getPosition(pos, destOri, left, right, i));
          out[i].setOri(getOrientation(destOri));
          complete[i]=false;                        // Not done with next phase
        }                                   // for (int i=0; i<out.size(); ++i)
        turn_to_dest.setActive(false);      // No longer turning to destination
        move_to_dest.setActive(false);           // Not moving to destination
        turn_to_heading.setActive(true);    // Start turning to final heading
      }                                     // if (phaseDone(in.getSource())
      else                  // if we're not yet done turing to final heading
        for (uint i=0; i<out.size(); ++i) out[i].setTX(false);  // Don't send
    }   // node:movementComplete[MovementComplete:in][MoveTo[5]=>(tanks[@];)]
  }                                                       // mode:move_to_dest

  mode:turn_to_heading
  {                                                    // mode:turn_to_heading
    node:movementComplete[MovementComplete:in]                   // One is done
                          [MovementComplete:out=>(parent;)]         // All done
    {     // node:movementComplete[MovementComplete:in][MovementComplete:out]
      if (phaseDone(in.getSource()))        // If we can go to the next phase
      {
        ori = destOri;                // We're at our destination orientation
        startTime = -1.0;                              // Get the current time
        turn_to_dest.setActive(false);      // No longer turning to destination
        move_to_dest.setActive(false);           // Not moving to destination
        turn_to_heading.setActive(false);     // Not turning to final heading
        for (uint i=0; i<complete.size(); ++i) complete[i]=false;
      }                                     // if (phaseDone(in.getSource())
      else                  // if we're not yet done turing to final heading
        out.setTX(false);                 // Don't send the final confirmation
    }   // node:movementComplete[MovementComplete:in][MoveTo[5]=>(tanks[@];)]
  }                                                    // mode:turn_to_heading
 }                                                           // process:Platoon
}
```

C.1.27. RedCompany.proc

```
{import process {Company} }
{import message {StartSimulation, SetColor, SetFormation, SetLinearPosition,
                MoveFormation, UnitSetup} }
{import spt {sptEnvironmentObject} }
{import gvm {gvmTank} }

{
  process:RedCompany(Company)
  {                                               // process:RedCompany(Company)
    method:init(public; void;)
    {                                              // method:init(public; void;)
      Company::init();                // Initialize the parent construct first
      view->setTitle("Red Tactical View");                          // Red view
      force = RED;                          // This has force designator "RED"
      view->setPosition(1032,27);            // Set the position of the window
      view->setSize(563, 516);                                   // Set the size
    }                                              // method:init(public; void;)

    mode:startup
    {                                                           // mode:startup
      node:start[StartSimulation:in]                          // Upon startup
              [SetColor:sc=>(platoons; cp;), // Set the color of the objects
               SetFormation:sf[]=>(platoons[@];),    // Set formation params
               SetLinearPosition:slp=>(cp;),        // Command post position
               MoveFormation:mf[]=>(platoons[@];):(10.0+@*5.0)]
      {                               // node:start[StartSimulation:in][ ... ]
        sc.set(1.0, 0.0, 0.0, 1.0);      // Set the color of subordinates to red
        double r = sqrt(5000.0);                       // Range between tanks
        double d = sqrt(2000000.0);              // Range between destinations
        for (uint i=0; i<platoons.size(); ++i)        // Loop over the platoons
        {
          double x = -8500-r*((double) i);                           // Location
          sf.push_back(me);                                 // Add a new message
          sf[i].setPos(x,x);                            // Set the tank position
          sf[i].setOri(r,r);         // Set the platoon orientation & spacing
          sf[i].setForm(V_FORMATION);                         // Use V formation

          x=d*(((double) i)-((double) (platoons.size()-1))/2.0);    // Dest loc
          mf.push_back(me);                                 // Add a new message
          mf[i].setPos(x, -x);                              // Set the destination
          mf[i].setOri(r,r);                                // Set the orientation
          mf[i].setForm(V_FORMATION);                       // Use the V formation
        }                               // for (uint i=0; i<platoons.size(); ++i)
        slp.set(-9500.0, -9500.0, 0.0); // Set the position of the command post
      }                               // node:start[StartSimulation:in][ ... ]
    }                                                           // mode:startup
  }                                               // process:RedCompany(Company)
}
```

C.1.28. RegisterEnvironmentObject.msg

```
{import spt {sptNewtonianMotion, sptLinearMotion, sptAngularMotion} }

{
  message:RegisterEnvironmentObject
  {                                         // message:RegisterEnvironmentObject
    double:r;                                  // Effective radius of the sensor
    ulong:f; // Which force does track belong to NEUTRAL (0), RED (1), BLUE (2)
    spt::NewtonianMotion:motion;                              // Motion paramters

    method:setRadius(public; void; double:R;) { r=R; } // Set the sensor radius
    method:setForce(public; void; ulong:F;) { f = F; }   // Set the force value
    method:getMotion(public; spt::NewtonianMotion;) { return motion; }

    method:getForce(public; ulong;) { return f; }      // Return the force index
    method:getRadius(public; double;) { return r; } // Return the sensor radius
    method:setMotion(public; void; spt::NewtonianMotion&:m;) { motion=m; }
  }                                         // message:RegisterEnvironmentObject
}
```

C.1.29. ScheduleAddTrack.msg

```
{import message {ScheduleTrackEvent} }
{message:ScheduleAddTrack(ScheduleTrackEvent);}
```

C.1.30. ScheduleLoseTrack.msg

```
{import message {ScheduleTrackEvent} }
{message:ScheduleLoseTrack(ScheduleTrackEvent);}
```

C.1.31. ScheduleTrackEvent.msg

```
{
  message:ScheduleTrackEvent
  {
    ulong:sensor;
    ulong:track;
    method:set(public; void; ulong:s; ulong:t;) { sensor=s; track=t; }
    method:getSensor(public; ulong;) { return sensor; }
    method:getTrack(public; ulong;) { return track; }
  }
}
```

C.1.32. SensorTrack.proc

```
{import process {NewtonianMotion} }
{import message {AddTrack, ChangeTrack, LoseTrack, AddEnvironment,
                 SetEnvironment, SetNewtonianMotion, Impact, Destroyed} }
{import spt {sptEnvironmentObject, sptNewtonianMotion} }
{import std {<vector>} }

{
  process:SensorTrack(NewtonianMotion)
  {                                      // process:SensorTrack(NewtonianMotion)
    ulong:envIndex((ulong) (-1));                // Index within the environment
    process:environment;                                  // Environment process
    process:parent;                           // Parent object to report back to
    double:radius(2000.0);                                      // Sensor Radius
    ulong:force(NEUTRAL);              // Initially neutral, until we know better
    spt::NewtonianMotion:tracks[];                 // Collection of known tracks
    std::set<ulong>:active;                       // Collection of active tracks

    method:isActive(public; bool; ulong:i;)                 // Is track i active?
    { return active.find(i)!=active.end(); }

    method:notify(public; void; std::vector<SetNewtonianMotion>&:out;)
    {     // method:notify(public; void; std::vector<SetNewtonianMotion>&:out;)
      NewtonianMotion::notify(out);                 // Call the parent version
      if (envIndex != ((ulong) -1))   // If we have registered with environment
      {
        out.push_back(me);                            // Allocate a new message
        out.back().addDest(environment);             // Add environment as dest
        out.back().addDest(parent);                  // Add environment as dest
        out.back().index = envIndex;                        // Specify the index
        out.back().set(nm);              // Specify the Linear Motion paramters
      }                                          // if (envIndex != ((ulong) -1))
    }     // method:notify(public; void; std::vector<SetNewtonianMotion>&:out;)

    mode:Default
    {                                                              // mode:Default
      node:addEnvironment[AddEnvironment:in]
                        [SetNewtonianMotion:out=>(parent;)]
      {                       // Node:addEnvironment[AddEnvironment:in][...]
        envIndex = in.index;                       // Save the environment index
        out.index = envIndex;                      // Notify parent of new index
        out.set(nm);                                // Set the motion parameters
      }                       // Node:addEnvironment[AddEnvironment:in][...]

      node:setEnvironment[SetEnvironment:in][]
      {                          // node:setEnvironment[SetEnvironment:in][]
        environment = in.environment;       // Environment in which this exists
        parent = in.getSource();              // Save the parent process handle
      }                          // node:setEnvironment[SetEnvironment:in][]

      node:addTrack[AddTrack:in][AddTrack:out=>(parent;)]
      {                          // node:addTrack[AddTrack:in][AddTrack:out]
```

```
        if (tracks.size() <= in.getTrack())       // If tracks isn't big enough
          tracks.resize(in.getTrack()+1);              // Resize the track list

        tracks[in.getTrack()] = in.getMotion();     // Save the motion paramter
        active.insert(in.getTrack());                  // Add an active element
        out.setMotion(in.getMotion());            // Notify as to object motion
        out.set(in.getTrack(), in.getForce());         // Provide force tracking
      }                              // node:addTrack[AddTrack:in][AddTrack:out]

      node:changeTrack[ChangeTrack:in][ChangeTrack:out=>(parent;)]
      {                    // node:changeTrack[ChangeTrack:in][ChangeTrack:out]
        tracks[in.getTrack()] = in.getMotion();     // Save the motion paramter
        out.setMotion(in.getMotion());            // Notify as to object motion
        out.set(in.getTrack(), in.getForce());         // Provide force tracking
      }                    // node:changeTrack[ChangeTrack:in][ChangeTrack:out]

      node:loseTrack[LoseTrack:in][LoseTrack:out=>(parent;)]
      {                          // node:loseTrack[LoseTrack:in][LoseTrack:out]
        active.erase(in.getTrack());                 // Track is no longer active
        out.set(in.getTrack(),in.getForce());      // Provide force information
      }                          // node:loseTrack[LoseTrack:in][LoseTrack:out]

      node:impact[Impact:in]                                  // We've been hit
                 [Destroyed:out[],       // Notify processes of our destruction
                  LoseTrack:lt[]=>(parent;),               // Lost all the tracks
                  SetNewtonianMotion:snm[]]   // Notify of new newtonian motion
      {                             // node:impact[Impact:in][Destroyed:out[]]
        nm.la(0.0, 0.0, 0.0, getTime());           // Stop linear acceleration
        nm.lv(0.0, 0.0, 0.0, getTime());                 // Stop linear motion
        nm.aa(0.0, 0.0, 0.0, getTime());          // Stop angular acceleration
        nm.av(0.0, 0.0, 0.0, getTime());                // Stop angular motion
        notify(snm);    // Notify views/environments about new newtonian motion

        std::map<process, gvm::object_index>::iterator i;      // For loop index
        for (i=views.begin(); i!=views.end(); ++i)        // Loop over index map
        {
          out.push_back(me);                           // Allocate a new message
          out.back().addDest(i->first);       // Add this view as a destination
          out.back().index = i->second;                     // Specify the index
        }                            // for (i=views.begin(); i!=views.end(); ++i)

        if (envIndex != ((ulong) -1)) // If we have registered with environment
        {
          out.push_back(me);                           // Allocate a new message
          out.back().addDest(environment);           // Add environment as dest
          out.back().addDest(parent);                // Add environment as dest
          out.back().index = envIndex;                      // Specify the index
        }                                     // if (envIndex != ((ulong) -1))

        std::set<ulong>::iterator t;                  // Index for active tracks
        for (t=active.begin(); t!=active.end(); ++t) // Loop over active tracks
        {
          lt.push_back(me);                   // Create a new LoseTrack message
          lt.back().set(*t, (force==RED ? BLUE : RED));  // tell of lost tracks
        }                        // for (t=active.begin(); t!=active.end(); ++t)
      }                              // node:impact[Impact:in][Destroyed:out[]]
    }                                                           // mode:Default
  }                                     // process:SensorTrack(NewtonianMotion)
}
```

C.1.33. SetAngularAcceleration.msg

```
{import message {SetMotion} }
{message:SetAngularAcceleration(SetMotion);}
```

C.1.34. SetAngularPosition.msg

```
{import message {SetMotion} }
{message:SetAngularPosition(SetMotion);}
```

C.1.35. SetAngularVelocity.msg

```
{import message {SetMotion} }
{message:SetAngularVelocity(SetMotion);}
```

C.1.36. SetEnvironment.msg

```
{
  message:SetEnvironment
  {                                                     // message:SetEnvironment
    process:environment;                // Reference to the environment process
    process:renderNode;                          // Root node in the scene graph

    method:setEnvironment(public; void; process:e;) { environment=e; }
    method:setNode(public; void; process:n;) { renderNode=n; }

    method:getEnvironment(public; process;) { return environment; }
    method:getNode(public; process;) { return renderNode; }
  }                                                     // message:SetEnvironment
}
```

C.1.37. SetFormation.msg

```
{import message {AdjustFormation} }
{message:SetFormation(AdjustFormation);}
```

C.1.38. SetLinearAcceleration.msg

```
{import message {SetMotion} }
{message:SetLinearAcceleration(SetMotion);}
```

C.1.39. SetLinearPosition.msg

```
{import message {SetMotion} }
{message:SetLinearPosition(SetMotion);}
```

C.1.40. SetLinearVelocity.msg

```
{import message {SetMotion} }
{message:SetLinearVelocity(SetMotion);}
```

C.1.41. SetMotion.msg

```
{import message {SetValue} }
{import spt {sptDefs} }

{
  message:SetMotion(SetValue)
  {                                                  // message:SetMotion(SetValue)
    double:t(getTime());              // Effective time of the motion paramters
    spt::vertex:v(0.0, 3);                          // Where the vector is stored

    method:set(public; void; double:x; double:y; double:z;)
    { v[0]=x; v[1]=y; v[2]=z; }

    method:set(public; void; spt::vertex:V;) { v = V; }
    method:setT(public; void; double:T;) { t = T; }             // Set the time
    method:get(public; spt::vertex;) { return v; }                 // Get value
    method:get(public; double; ulong:i;) { return (i<v.size() ? v[i] : 0.0); }
    method:getT(public; double;) { return t; }                  // Get the time
  }                                                  // message:SetMotion(SetValue)
}
```

C.1.42. SetNewtonianMotion.msg

```
{import message {SetValue} }
{import spt {sptNewtonianMotion, sptLinearMotion, sptAngularMotion} }

{
  message:SetNewtonianMotion(SetValue)
  {                                     // message:SetNewtonianMotion(SetValue)
    spt::NewtonianMotion:nm;                                  // Motion paramters

    method:set(public; void; spt::NewtonianMotion&:n;) { nm=n; }
    method:get(public; spt::NewtonianMotion;) { return nm; }
  }                                      // message:SetLinearVelocity(SetValue)
}
```

C.1.43. SetTankState.msg

```
{import message {SetValue} }

{
  message:SetTankState(SetValue)
  {                                           // message:SetTankState(SetValue)
    double:az(0.0);                                             // Gun azimuth
    double:el(0.0);                                           // Gun elevation
    double:azRate(0.0);                                   // Azimuth slew rate
    double:elRate(0.0);                                 // Elevation slew rate
    double:azStart(0.0);                            // Azimuth slew start time
    double:elStart(0.0);                          // Elevation slew start time
    double:azStop(0.0);                             // Azimuth slew start time
    double:elStop(0.0);                           // Elevation slew start time

    method:setAzimuth(public; void; double:a; double:r; double:s0; double:s1;)
      { az=a; azRate=r; azStart=s0; azStop=s1; }
    method:setElevation(public; void; double:e; double:r; double:s0;
                                      double:s1;)
      { el=e; elRate=r; elStart=s0; elStop=s1; }

    method:getAzimuth(public; double;) { return az; }
    method:getAzimuthRate(public; double;) { return azRate; }
    method:getAzimuthStart(public; double;) { return azStart; }
    method:getAzimuthStop(public; double;) { return azStop; }
    method:getElevation(public; double;) { return el; }
    method:getElevationRate(public; double;) { return elRate; }
    method:getElevationStart(public; double;) { return elStart; }
    method:getElevationStop(public; double;) { return elStop; }
  }                                           // message:SetTankState(SetValue)
}
```

C.1.44. Stop.msg

```
{message:Stop;}
```

C.1.45. StopAzimuthSlew.msg

```
{import message {StopSlew} }
{message:StopAzimuthSlew;}
```

C.1.46. StopElevationSlew.msg

```
{import message {StopSlew} }
{message:StopElevationSlew;}
```

C.1.47. StopSlew.msg

```
{message:StopSlew;}
```

C.1.48. Tank.proc

```
{import process {Vehicle, Munition} }
{import message {SetTankState, AddView, UnitSetup, RegisterEnvironmentObject,
                 SetEnvironment, AddTrack, ChangeTrack, LoseTrack, SetColor,
                 Fire, Attack, StopAzimuthSlew, StopElevationSlew, Hit,
                 Impact} }
{import spt {sptEnvironmentObject, sptNewtonianMotion, sptLinearMotion,
             sptAngularMotion} }
{import std {<map>, <queue>} }
{import {<math.h>} }
```

```
{
  process:Tank(Vehicle)
  {                                                   // process:Tank(Vehicle)
    Munition:rounds[50];             // We have fifty of them we can fire off
    double:azMaxRate(PI/4.0);                             // Azimuth slew rate
    double:elMaxRate(PI/8.0);                           // Elevation slew rate
    ulong:round(0);                                       // Next round to use
    double:mv(5000.0);                   // Set muzzle velocity to 5000 m/s

    double:az(0.0);                                             // Gun azimuth
    double:el(0.0);                                           // Gun elevation
    double:azRate(0.0);                                   // Azimuth slew rate
    double:elRate(0.0);                                 // Elevation slew rate
    double:azStart(0.0);                            // Azimuth slew start time
    double:elStart(0.0);                          // Elevation slew start time
    double:azStop(2.0*Clock::getEndTime());          // Azimuth slew stop time
    double:elStop(2.0*Clock::getEndTime());        // Elevation slew stop time
    std::queue<ulong>:targets;                              // Targets to shoot

    method:init(public; void;)
    {                                           // method:init(public; void;)
      maxVel = 40.0;                        // Set maximum velocity to 40 m/s
      maxRot = 0.5*PI;                      // Set max rotation angle to PI/2
      Vehicle::init();                                 // Call parent version
      attack.setActive(false);                    // Nothing to attack right now
    }                                           // method:init(public; void;)

    method:setGunPos(public; void; double:a; double:e;) { az=a; el=e; }

// method:angle returns the radian angle value of t in the range {-PI, PI]
    method:angle(public; double; double:t;)
    {                               // method:angle(public; double; double:t;)
      t = fmod(t, 2.0*PI);               // Get in range [0,2*PI) or (-2*PI, 0]

      if (t<=-PI) t+=2.0*PI;                        // Get in range (-2*PI, PI)
      else if (t>=PI) t-=2.0*PI;                       // Get in range (-PI, PI]
      return t;                                             // Return the value
    }                               // method:angle(public; double; double:t;)

// method:bearing returns the radian angle value of t in the range [0, 2.0*PI)
    method:bearing(public; double; double:t;)
    {                             // method:bearing(public; double; double:t;)
      t = fmod(t, 2.0*PI); if (t<0.0) t+=2.0*PI;       // Get in range [0, 2*PI)
      return t;                                             // Return the value
    }                             // method:bearing(public; double; double:t;)

    method:getAzimuth(public; double; double:t;)   // Effective time of azimuth
    {                            // method:getAzimuth(public; void; double:t;)
      double dt = min(azStop, t)-azStart;               // Get the elapsed time
      return angle(az+azRate*dt);                   // Get the current bearing
    }                            // method:getAzimuth(public; void; double:t;)

    method:getElevation(public; double; double:t;)  // Effective elevation time
    {                          // method:getElevation(public; void; double:t;)
      double dt = min(elStop, t)-elStart;               // Get the elapsed time
      return angle(el+elRate*dt);                 // Get the current elecation
    }                          // method:getElevation(public; void; double:t;)

    method:updateGunPos(public; void; double:t;)  // Update to current position
    {                          // method:updateGunPos(public; void; double:t;)
      az = getAzimuth(t);                     // Update the azimuth to time t
      el = getElevation(t);                 // Update the elevation to time t
      if (azStop < t)   // If we haven't actually arrived at the stop time, yet
      {
        azStop = 2.0*Clock::getEndTime(); // Specify end time of current motion
        azRate = 0.0;                                            // Stop motion
      }                                                      // if (azStop < t)

      if (elStop < t)   // If we haven't actually arrived at the stop time, yet
      {
```

```
    elStop = 2.0*Clock::getEndTime(); // Specify end time of current motion
    elRate = 0.0;                                            // Stop motion
  }                                                       // if (elStop < t)

  azStart = elStart = t;           // Specify time t as the new start time
}                            // method:updateGunPos(public; void; double:t;)

method:turnGunTo(public; void; double:a; double:e;)
{                       // method:turnGunTo(public; void; double:a; double:e)
  updateGunPos(getTime());                    // Get the current gun position
  azRate  = angle(a-az) < 0 ? -azMaxRate : azMaxRate;      // Turn direction
  elRate  = angle(e-el) < 0 ? -elMaxRate : elMaxRate;      // Turn direction
  azStop  = getTime()+(angle(a-az)/azRate);   // Get azimuth slew stop time
  elStop  = getTime()+(angle(e-el)/elRate); // Get elevation slew stop time
}                       // method:turnGunTo(public; void; double:a; double:e)

method:stopAzimuth(public; void; double:t;)       // Stop azimuth at time t
{                              // method:stopAzimuth(public; void; double:t;)
  az = getAzimuth(t);                              // Update the gun azimuth
  azRate = 0.0;                                     // Stop azimuth slewing
  azStart = t;                                        // New affective time
  azStop = 2.0*Clock::getEndTime();                    // Get the stop time
}                              // method:stopAzimuth(public; void; double:t;)

method:stopElevation(public; void; double:t;)   // Stop elevation at time t
{                            // method:stopElevation(public; void; double:t;)
  el = getElevation(t);                          // Update the gun elevation
  elRate = 0.0;                                   // Stop elevation slewing
  elStart = t;                                        // New affective time
  elStop = 2.0*Clock::getEndTime();                    // Get the stop time
}                            // method:stopElevation(public; void; double:t;)

method:notify(public; void; std::vector<SetTankState>&:out;)
{          // method:notify(public; void; std::vector<SetTankState>&:out;)
  std::map<process, gvm::object_index>::iterator i;       // For loop index
  for (i=views.begin(); i!=views.end(); ++i)         // Loop over index map
  {
    out.push_back(me);                              // Allocate a new message
    out.back().addDest(i->first);          // Add this view as a destination
    out.back().index = i->second;                      // Specify the index
    out.back().setAzimuth(az,azRate,azStart,azStop);      // Specify azimuth
    out.back().setElevation(el,elRate,elStart,elStop); // Specify elevation
  }                               // for (i=views.begin(); i!=views.end(); ++i)
}          // method:notify(public; void; std::vector<SetTankState>&:out;)

method:getPos(public; spt::vertex; spt::vertex:p; spt::vertex:v;
                                   spt::vertex:a; double:t;)
{ return p+(v+0.5*t*a)*t; } // Position affected by velocity & acceleration

method:getPos(public; spt::vertex; spt::vertex:p; spt::vertex:v; double:t;)
{ return p+(t*v); }                    // Position affected only by velocity

method:getPos(public; spt::vertex; double:a; double:e; double:t;)
{                                  // method:getPos(public;spt::vertex; ... )
  spt::vertex pos(0.0, 3);
  spt::vertex acc(0.0, 3);
  spt::vertex vel(0.0, 3);

  vel[0] = mv*cos(e)*cos(a);
  vel[1] = mv*cos(e)*sin(a);
  vel[2] = mv*sin(e);

  acc[0] = 0.0;
  acc[1] = 0.0;
  acc[2] = -9.8;

  return getPos(pos, vel, acc, t);
}                                  // method:getPos(public;spt::vertex; ... )

method:turnTime(public; double; double:a; double:e;)
{ return max(fabs(angle(a)/azMaxRate), fabs(angle(e)/elMaxRate)); }
```

```
method:getIntercept(public; void; spt::vertex:p; spt::vertex&:sln;)
{                // method:getIntercept(public; void; spt::vertex:p; ...)
  double r = norm(p);                                    // Range to target
  sln[0] = angle(atan2(p[1], p[0]));             // Get the gun's azimuth
  sln[1] = 0.5*asin(9.8*r/(mv*mv));            // Get the gun's elevation
  sln[2] = mv*sin(sln[1])/4.9;                     // Get the travel time
}                // method:getIntercept(public; void; spt::vertex:p; ...)

method:rotate(public; spt::vertex; spt::vertex&:v; double:theta;)
{              // method:rotate(public; spt::vertex; spt::vertex; double;)
  double ct=cos(theta), st=sin(theta);   // Get transformation coefficients
  spt::vertex rv(0.0,3);                 // Return value valarray
  rv[0]=ct*v[0]-st*v[1];                        // Get the new 'x' component
  rv[1]=st*v[0]+ct*v[1];                        // Get the new 'y' component
  return rv;                 // Return the valarray to the calling routine
}              // method:rotate(public; spt::vertex; spt::vertex; double;)

method:aim(public; void; ulong:track; double:err;)          // Aim at track
{                    // method:aim(public; void; ulong:track; double:err;)
  double tt=0;                                     // Turn time to target
  double t = getTime();                  // Convenience for the current time
  double a = getAzimuth(t), e = getElevation(t);   // Get current gun state
  double bearing = nm.ap(t)[2];             // Get the current tank bearing
  spt::vertex rp=tracks[track].lp(t)-nm.lp(t);             // Rel target pos
  spt::vertex rv=tracks[track].lv(t)-nm.lv(t);             // Rel target vel
  rp = rotate(rp,-bearing);          // Rotate to reflect a relative bearing
  rv = rotate(rv,-bearing);          // Rotate this to relative bearing, too

  spt::vertex diff(0.0,3);              // Vector between impact and target
  spt::vertex sln(0.0,3);             // Place to hold gun firing solution
  ulong c=0;                            // Counter to avoid infinite loops
  t = 0;                        // Start with current time as a reference
  do   // Iteratively get solutions until within acceptable margin of error
  {
    getIntercept(getPos(rp,rv,sln[2]+tt), sln);     // Solution to position
    tt = turnTime(sln[0]-a, sln[1]-e);               // Calculate turn time
    diff=getPos(rp,rv,sln[2]+tt)-getPos(sln[0],sln[1],sln[2]);  // Imp diff
  }                                                                   // do
  while (norm(diff)>err && ++c<10);   // Loop until solution, or divergence
  turnGunTo(sln[0], sln[1]);   // Start turning the gun to where it belongs
}                    // method:aim(public; void; ulong:track; double:err;)

mode:Default
{                                                          // mode:Default
  node:addView[AddView:in][SetTankState:out=>(in.getSource();)]
  {                    // node:addView[AddView:in][SetTankState:out]
    out.setAzimuth(az, azRate, azStart, azStop);      // Specify gun azimuth
    out.setElevation(el, elRate, elStart, elStop); // Specify gun elevation
    out.index = in.index;                             // Set the index value
  }                    // node:addView[AddView:in][SetTankState:out]

  node:unitSetup[UnitSetup:in]                   // Setting the environment
               [SetEnvironment:se=>(rounds;),             // Establish env.
                SetColor:sc=>(me;),              // Set this unit's color
                RegisterEnvironmentObject:rst=>(in.environment;)]    // Reg
  {              // node:unitSetup[UnitSetup:in][SetEnvironment:se]
    se.setEnvironment(in.getEnvironment());          // Set the environment
    se.setNode(in.getNode());                     // Set the rendering node
    rst.setForce(force = in.getForce());         // Set the force component
    rst.setRadius(radius = in.getRadius());        // Set the sensor radius
    rst.setMotion(nm);              // Set the newtonian motion parameters
    if (force==RED) sc.set(1.0, 0.0, 0.0);         // Set this unit's color
    else if (force==BLUE) sc.set(0.0, 0.0, 1.0);             // Red or Blue
  }              // node:unitSetup[UnitSetup:in][SetEnvironment:se]

  node:addTrack[AddTrack:in]             // Upon notification of a new track
             [Attack:out=>(me;)]                   // Attack the new track
  {                        // node:addTrack[AddTrack:in][Attack:out]
    attack.setActive(true);                     // Start the attack sequence
    out.setTX(targets.empty());   // Don't bother if we're already attacking
    targets.push(in.getTrack());     //
  }                                 // node:addTrack[AddTrack:in][Attack:out]

  node:impact[Impact:in][]
  {                                                 // node:impact[Impact:in][]
    Default.setActive(false);
    attack.setActive(false);
    turn_to_dest.setActive(false);
    move_to_dest.setActive(false);
    turn_to_heading.setActive(false);
  }                                                 // node:impact[Impact:in][]
}                                                                // mode:Default

mode:attack
{                                                                // mode:attack
  node:attack[Attack:in]                  // Upon notification of a new track
            [StopAzimuthSlew:sas=>(me;):(azStop),          // Slew azimuth
             StopElevationSlew:ses=>(me;):(elStop),       // Slew elevation
             SetTankState:sts[]]          // Notify views of new tank state
  { // node:addTrack[AddTrack:in][StopAzimuthSlew, StopElevationSlew, ... ]
    if (round<rounds.size())              // If we have some tank rounds left
    {
      aim(targets.front(),0.01);      // Start moving  gun to aim at target
      notify(sts); // Notify the views that the gun parameters have changed
    }                                              //if (round<rounds.size())
    else                          // If there are no more tank rounds left
    {
      sas.setTX(false);                   // Don't bother realigning the gun
      ses.setTX(false);                          // Or changing its elevation
      attack.setActive(false);                   // Turn the attack mode off
    }                                 // else from if (round<rounds.size())
  } // node:addTrack[AddTrack:in][StopAzimuthSlew, StopElevationSlew, ... ]

  node:StopAzimuth[StopAzimuthSlew:in]     // When the azimuth stops slewing
                 [Fire:f,       // Fire the gun if we're all done aiming it
                  SetTankState:sts[]]     // Notify views of new tank state
  {     // node:StopAzimuth[StopAzimuthSlew:in][Fire:f, SetTankState:sts[]]
    double t = getTime();                             // Get the current time
    if (t == azStop)                  // If this message is for current slew
    {
      stopAzimuth(t);                             // Stop slewing the azimuth
      notify(sts);          // Notify the views that the azimuth has stopped
      if (elStop>Clock::getEndTime())       // If the elevation slew stopped
      {
        f.set(mv,az+nm.ap(t)[2],el,nm.lp(t),nm.lv(t)); // Set firing params
        f.addDest(rounds[round++]);        // Tell it to the next tank round
      }                                   // if (elStop>Clock::getEndTime())
      else f.setTX(false);    // Don't fire until elevation slew is complete
    }                                             // if (getTime() == azStop)
  }     // node:StopAzimuth[StopAzimuthSlew:in][Fire:f, SetTankState:sts[]]

  node:StopElevation[StopElevationSlew:in]  // When elevation stops slewing
                   [Fire:f,     // Fire the gun if we're all done aiming it
                    SetTankState:sts[]]  // Notify views of new tank state
  {  // node:StopElevation[StopElevationSlew:in][Fire:f,SetTankState:sts[]]
    double t = getTime();                             // Get the current time
    if (t == elStop)                  // If this message is for current slew
    {
      stopElevation(t);                         // Stop slewing the elevation
      notify(sts);        // Notify the views that the elevation has stopped
      if (azStop>Clock::getEndTime())       // If the elevation slew stopped
      {
        f.set(mv,az+nm.ap(t)[2],el,nm.lp(t),nm.lv(t)); // Set firing params
        f.addDest(rounds[round++]);        // Tell it to the next tank round
      }                                   // if (elStop>Clock::getEndTime())
      else f.setTX(false);    // Don't fire until elevation slew is complete
    }                                             // if (getTime() == elStop)
  }  // node:StopElevation[StopElevationSlew:in][Fire:f,SetTankState:sts[]]

  node:changeTrack[ChangeTrack:in][Attack:out]
  {                           // node:changeTrack[ChangeTrack:in][Attack:out]
```

```
        if (in.getTrack() == targets.front())       // If this is the target...
          out.set(in.getTrack());                        // Reaim and start again
        else                                // If it's not the one we're aiming at
          out.setTX(false);                              // Don't change anything
      }                          // node:changeTrack[ChangeTrack:in][Attack:out]

      node:hit[Hit:in]
              [Attack:out]
      {                                             // node:hit[Hit:in][Attack:out]
        std::set<ulong> hits;                                         // Sorted hits

        for (ulong i=0; i<in.track.size(); ++i)            // Loop over the hits
          hits.insert(in.track[i]);         // Put them in the set to sort them

        while (!targets.empty() &&                        // While targets remain
               (hits.find(targets.front())!=hits.end() ||  // but they were hit
               active.find(targets.front())==active.end()))          // or lost
          targets.pop();                         // Pop the target from the queue

        if (!targets.empty())                                // If targets remain
          out.set(targets.front());                        // Attack the next one
        else                                              // If no targets remain
        {
          out.setTX(false);                      // Don't send the attack message
          attack.setActive(false);                    // Turn off the attack mode
        }          // else from if (!targets.empty()) out.set(targets.front()))
      }                                             // node:hit[Hit:in][Attack:out]
    }                                                             // mode:attack
  }                                                     // process:Tank(Vehicle)
}
```

C.1.49. TrackEvent.msg

```
{
  message:TrackEvent
  {                                                       // message:TrackEvent
    ulong:force(0);                           // Force identifier for the track
    ulong:track( ((ulong) -1) );     // Index of track being detected by sensor

    method:set(public; void; ulong:t; ulong:f;) { track=t; force=f; }
    method:setTrack(public; void; ulong:t;) { track=t; }
    method:setForce(public; void; ulong:f;) { force=f; }
    method:getForce(public; ulong;) { return force; }
    method:getTrack(public; ulong;) { return track; }
  }                                                       // message:TrackEvent
}
```

C.1.50. TrackMotionEvent.msg

```
{import message {TrackEvent} }
{import spt {sptNewtonianMotion} }

{
  message:TrackMotionEvent(TrackEvent)
  {                                   // message:TrackMotionEvent(TrackEvent)
    spt::NewtonianMotion:motion;            // Motion parameters of the track

    method:setMotion(public; void; spt::NewtonianMotion:m;)
      { motion=m; motion.setStopTime(2.0*Clock::getEndTime()); }
    method:getMotion(public; spt::NewtonianMotion;) { return motion; }
  }                                   // message:TrackMotionEvent(TrackEvent)
}
```

C.1.51. UnitSetup.msg

```
{import message {SetEnvironment} }

{
  message:UnitSetup(SetEnvironment)
  {
    ulong:force;                 // Force identifier for the destination object
    double:radius;                  // Radius of detection for destination unit

    method:set(public; void; ulong:f; double:r; process:e; process:n;)
    { force=f; radius=r; environment=e; renderNode=n; }

    method:setForce(public; void; ulong:f;) { force=f; }
    method:setRadius(public; void; double:r;) { radius=r; }

    method:getForce(public; ulong;) { return force; }
    method:getRadius(public; double;) { return radius; }
  }
}
```

C.1.52. Vehicle.proc

```
{import process {SensorTrack} }
{import message {MoveTo, Stop, MovementComplete, HoldPosition,
                 SetNewtonianMotion} }
{import std {<valarray>} }
{import spt {sptDefs} }
{import {<math.h>} }

{
  process:Vehicle(SensorTrack)
  {                                              // process:Vehicle(SensorTrack)
    double:maxVel;                                        // Max vehicle velocity
    double:maxRot;                                   // Max vehicle rotation rate
    spt::vertex:destPos(3);                               // Destination position
    spt::vertex:destOri(3);                            // Destination orientation
    double:orderTime;  // Last order time, to ignore obsolete movement messages

    method:init(public; void;)
    {                                               // method:init(public; void;)
      turn_to_dest.setActive(false);                // Now turning to destination
      move_to_dest.setActive(false);             // Not yet moving to destination
      turn_to_heading.setActive(false);      // Not turning to final heading yet
    }                                               // method:init(public; void;)

    method:halt(protected; void; double:st; double:et;)
    {                    // method:halt(protected; void; double:st; double:et;)
      spt::vertex z(0.0, 3);                                       // z for zero
      nm.set(nm.lp(st), z, z, nm.ap(st), z, z, st, et);
      turn_to_dest.setActive(false);                // Now turning to destination
      move_to_dest.setActive(false);             // Not yet moving to destination
      turn_to_heading.setActive(false);      // Not turning to final heading yet
    }                    // method:halt(protected; void; double:st; double:et;)

    method:turn(protected; void; double:r; double:st; double:et;)   // Rotation
    {         // method:turn(protected; void; double:r; double:st; double:et;)
      spt::vertex z(0.0, 3);                                       // z for zero
      spt::vertex tr(0.0, 3);                                       // Turn rate
      tr[2]=r;                                                // Set the turn rate
      nm.set(nm.lp(st),z,z,nm.ap(st),tr,z,st,et);
    }                                    // method:turn(protected; void; double:r;)

    method:forward(protected; void; double:r; double:st; double:et;)  // Rate
    {                                  // method:forward(protected; void; double:r;)
      spt::vertex co(nm.ap(st));                        // Get current orientation
      spt::vertex z(0.0, 3);                                       // z for zero
      spt::vertex v(0.0, 3);                                  // Velocity vector
      v[0] = r*cos(co[2]);               // Get the velocity in the x direction
      v[1] = r*sin(co[2]);               // Get the velocity in the y direction

      nm.set(nm.lp(st), v, z, co, z, z, st, et);
      turn_to_dest.setActive(false);           // No need to turn to destination
      move_to_dest.setActive(true);                    // Moving to destination
      turn_to_heading.setActive(false);         // Not turning to final heading
    }                                  // method:forward(protected; void; double:r;)

    mode:Default
```

```
{                                                          // mode:Default
  node:holdPosition[HoldPosition:in]
                   [SetNewtonianMotion:out[]]
  {         // node:holdPosition[HoldPosition:in][SetNewtonianMotion:out[]]
    halt(getTime(), 2.0*Clock::getEndTime());            // Stop all motion
    notify(out);                            // Generate the output messages
  }         // node:holdPosition[HoldPosition:in][SetNewtonianMotion:out[]]

  node:moveTo[MoveTo:in]         // Receive order to move to a position/ori
             [Stop:st=>(me;),                               // Stop turning
              SetNewtonianMotion:out[],       // Inform views & environment
              MovementComplete:mc=>(parent;)]       // Movement is complete
  {         // node:moveTo[MoveTo:in][StopTurn:st,SetNewtonianMotion:out[]]
    destPos = in.getPos();                 // Save the destination position
    destOri = in.getOri();              // Save the destination orientation
    spt::vertex cp(nm.lp(getTime()));               // Get current position
    spt::vertex co(nm.ap(getTime()));            // Get current orientation
    spt::vertex rp(destPos-cp);                        // Relative position
    double dist=norm(rp);           // Get relative distance to destination
    double theta = dist>0 ? atan2(rp[1], rp[0])
                          : co[2];        // Direction to dest if not there
    if (theta<0.0) theta += PI*2.0;                         // All positive
    double rb=spt::AngularMotion::angDiff(theta,co[2]);  // Get rel bearing
    double rh=spt::AngularMotion::angDiff(destOri[2],co[2]); // Rel heading
    orderTime = getTime();                     // Reacting to current order
    mc.setTX(false);                          // Don't transmit, by default

    if (dist==0.0 && rh==0.0)                              // Already there?
    {
      mc.setTX(true);                           // Inform parent we're done
      st.setTX(false);                       // Don't transmit stop message
      halt(getTime(), 2.0*Clock::getEndTime());          // Stop all motion
    }                                         // If we're at the destination
    else if (dist==0.0)                      // If all we need to do is turn
    {
      double stop = orderTime+fabs(rh/maxRot);  // Get stop time for motion
      double rate = (rh<0.0 ? -maxRot : maxRot);    // Get proper turn rate
      st.setTime(stop);                            // Get the rotation time
      turn(rate, orderTime, stop);             // Set the turning parameters
      turn_to_dest.setActive(false);          // Done turning to destination
      move_to_dest.setActive(false);           // Done moving to destination
      turn_to_heading.setActive(true);      // Turning to final heading yet
    }                                                 // else if (dist==0.0)
    else if (rb==0.0)                   // If pointed in the right direction
    {
      double stop = orderTime+fabs(dist/maxVel);    // Get motion stop time
      st.setTime(stop);                            // Get the rotation time
      forward(maxVel, orderTime, stop); // Move forward at the max velocity
    }                                                    // If (dist != 0.0)
    else                           // If we need to turn to the destination
    {
      double stop = orderTime+fabs(rb/maxRot);  // Get stop time for motion
      double rate = (rb<0.0 ? -maxRot : maxRot);    // Get proper turn rate
      st.setTime(stop);                            // Get the rotation time
      turn(rate, orderTime, stop);            // Set the turning parameters
      turn_to_dest.setActive(true);           // Now turning to destination
      move_to_dest.setActive(false);       // Not yet moving to destination
      turn_to_heading.setActive(false); // Not turning to final heading yet
    }
    notify(out);                            // Generate the output messages
  }            // node:moveTo[MoveTo:in][Stop:st,SetNewtonianMotion:out[]]
}                                                           // mode:Default

mode:turn_to_dest
{                                                      // mode:turn_to_dest
  node:stop[Stop:in]             // Receive order to move to a position/ori
           [Stop:st=>(me;),                                 // Stop turning
            SetNewtonianMotion:out[],         // Inform views & environment
            MovementComplete:mc=>(parent;)]         // Movement is complete
  {         // node:stop[Stop:in][Stop,SetNewtonianMotion,MovementComplete]
    mc.setTX(false);                                 // Probably not done yet
```

```
        if (in.getGenTime()==orderTime)          // If this is something we obey
        {
          spt::vertex cp(nm.lp(getTime()));              // Get current position
          spt::vertex co(nm.ap(getTime()));                   // Get current ori
          spt::vertex rp(destPos-cp);                       // Relative position
          double dist=norm(rp);                       // Distance to destination
          orderTime = getTime();                    // Reacting to current order
          double stop = orderTime+fabs(dist/maxVel);     // Get motion stop time
          st.setTime(stop);                             // Get the rotation time
          forward(maxVel, orderTime, stop); // Move forward at the max velocity
          notify(out);                          // Generate the output messages
        }                                     // if (in.getGenTime()==orderTime)
        else           // If the order was from something we implicitly revoked
          st.setTX(false);
      }          // node:stop[Stop:in][Stop,SetNewtonianMotion,MovementComplete]
    }                                                       // mode:turn_to_dest

    mode:move_to_dest
    {                                                       // mode:move_to_dest
      node:stop[Stop:in]             // Receive order to move to a position/ori
               [Stop:st=>(me;),                                 // Stop turning
                SetNewtonianMotion:out[],          // Inform views & environment
                MovementComplete:mc=>(parent;)]          // Movement is complete
      {          // node:stop[Stop:in][Stop,SetNewtonianMotion,MovementComplete]
        mc.setTX(false);                                // Probably not done yet
        if (in.getGenTime()==orderTime)          // If this is something we obey
        {
          spt::vertex co(nm.ap(getTime()));          // Get current ori
          double rh=spt::AngularMotion::angDiff(destOri[2],co[2]);  // Rel hdng
          orderTime = getTime();                    // Reacting to current order

          if (rh != 0.0)                                      // Some rotation?
          {
            double stop = orderTime+fabs(rh/maxRot);     // Get motion stop time
            double rate = (rh<0.0 ? -maxRot : maxRot);  // Get proper turn rate
            st.setTime(stop);                           // Get the rotation time
            turn(rate, orderTime, stop);            // Set the turning parameters
            turn_to_dest.setActive(false);        // Done turning to destination
            move_to_dest.setActive(false);         // Done moving to destination
            turn_to_heading.setActive(true);     // Turning to final heading yet
          }                                               // else if (rh != 0.0)
          else                          // If already in position & orientation
          {
            mc.setTX(true);                             // Inform parent we're done
            st.setTX(false);                      // Don't transmit stop message
            halt(getTime(), 2.0*Clock::getEndTime());           // Stop all motion
          }
          notify(out);                          // Generate the output messages
        }                                     // if (in.getGenTime()==orderTime)
        else           // If the order was from something we implicitly revoked
          st.setTX(false);
      }          // node:stop[Stop:in][Stop,SetNewtonianMotion,MovementComplete]
    }                                                       // mode:move_to_dest

    mode:turn_to_heading
    {                                                    // mode:turn_to_heading
      node:stop[Stop:in]             // Receive order to move to a position/ori
               [SetNewtonianMotion:out[],          // Inform views & environment
                MovementComplete:mc=>(parent;)]          // Movement is complete
      {               // node:stop[Stop:in][SetNewtonianMotion,MovementComplete]
        if (in.getGenTime()==orderTime)          // If this is something we obey
        {
          halt(getTime(), 2.0*Clock::getEndTime());             // Stop all motion
          notify(out);                          // Generate the output messages
        }                                     // if (in.getGenTime()==orderTime)
        else           // If the order was from something we implicitly revoked
          mc.setTX(false);
      }               // node:stop[Stop:in][SetNewtonianMotion,MovementComplete]
    }                                                    // mode:turn_to_heading
  }                                             // process:Vehicle(SensorTrack)
}
```

C.1.53. gvm/gvmAddTrack.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmAddTrack.h - Class declaration for the gvm::AddTrack class
////////////////////////////////////////////////////////////////////////////////

#ifndef ADDTRACK_H_INCLUDED
#define ADDTRACK_H_INCLUDED

#include "gvmChangeTrack.h"
#include "gvmObject.h"

#define GVM_AddTrack GVM_UserMessage003

namespace gvm
{                                                               // namespace gvm
  class View;                    // Forward declaration of the gvm::View class

  class AddTrack : public ChangeTrack
  {                                        // class AddTrack : public ChangeTrack
    private:
      double radius;                                          // Radius of tank
      ulong force;                                   // Set the track iff value

    public:
      AddTrack(View&, double, object_index, const spt::NewtonianMotion&,
               GLdouble, ulong);

      virtual void send(void);                    // Deliver the message payload

  };                                          // class AddTrack : public Message
}                                                               // namespace gvm

#endif
```

C.1.54. gvm/gvmAddTrack.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmAddTrack.cxx - Class method definitions for the gvm::AddTrack
//                       class
////////////////////////////////////////////////////////////////////////////////

#include "gvmTacticalView.h"
#include "gvmAddTrack.h"
#include "gvmView.h"

namespace gvm
{                                                               // namespace gvm
  AddTrack::AddTrack(View& v,
                     double t,
                     object_index i,
                     const spt::NewtonianMotion& nm,
                     double r,
                     ulong f)
    : ChangeTrack(v, t, GVM_AddTrack, i, nm), radius(r), force(f)
  {  // AddTrack::AddTrack(View&,double,object_index,spt::NewtonianMotion, ...)
  }  // AddTrack::AddTrack(View&,double,object_index,spt::NewtonianMotion, ...)

  void AddTrack::send(void)
  {                                               // void AddTrack::send(void)
    gvm::TacticalView& v = dynamic_cast<gvm::TacticalView&>(getView());
    v.addTrack(getDest(), nm, radius, force);
  }                                               // void AddTrack::send(void)
}                                                               // namespace gvm
```

C.1.55. gvm/gvmBattleView.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmBattleView.h - Defines the gvm::BattleView class in which all gvm::Object
//                   instances are viewed.
////////////////////////////////////////////////////////////////////////////////
```

```
#ifndef GVMBATTLEVIEW_H_INCLUDED
#define GVMBATTLEVIEW_H_INCLUDED

#include <string>
#include <iostream>
#include "gvmView3D.h"
#include "gvmTank.h"
#include "spt/sptDefs.h"

namespace gvm
{                                                                  // namespace gvm
  class BattleView : public View3D
  {                                              // class BattleView : public View
    protected:
      spt::vertex vnv;                                        // View normal vector
      spt::vertex vup;                                            // View up vector
      spt::vertex vrv;                                        // Right side of view
      double forward, up, right;            // Current speed in these directions
      double speed;                           // Speed of motion when in motion

    public:
      BattleView(void);                                        // Class constructor

      virtual void createObject(object_handle);     // Schedule object creation
      virtual void updateTravel(void);        // Update camera motion parameters
      virtual void begin(void);               // Start rendering the scene graph
      virtual bool isDisplay(void);            // Should we update the display?
      virtual void keydown(byte,int,int);           // Key press event callback
      virtual void keyup(byte,int,int);           // Key release event callback
      virtual void motion(int, int);            // Active mouse motion callback
  };                                             // class BattleView : public View
}                                                                  // namespace gvm

#endif
```

C.1.56. gvm/gvmBattleView.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmBattleView.cxx - Class member and static definitions of the
//                     gvm::BattleView class.
////////////////////////////////////////////////////////////////////////////////

#include "Exception.h"
#include "gvmTank.h"
#include "gvmCommandPost.h"
#include "gvmBattleView.h"
#include "gvmMunition.h"
#include "gvmGround.h"

namespace gvm
{                                                                  // namespace gvm
  BattleView::BattleView(void)                             // Window for the view
    : vnv(3), vup(3), vrv(3), forward(0.0), up(0.0), right(0.0), speed(1.0)
  {                                                 // BattleView::BattleView(void)
    zFar = 40000.0;                     // Set the far clipping plane to 40,000m
    pos[0] = 0.0; pos[1] = 0.0; pos[2] = 2000.0;
    ori[0] = 0.0; ori[1] = 0.0; ori[2] = 0.0;
    setTitle("BattleView");
    setSceneChange(true);                                  // Need to refresh display
    redisplay();                                         // Redisplay the environment
  }                                                 // BattleView::BattleView(void)

  void BattleView::createObject(object_handle h)
  {                                    // void BattleView::createObject(object_handle)
    if (h.second >= objectList.size())                // Is this lined up properly
      throw Exception::Nonspecific("Object count mis-alignment.");

    switch (h.first)                                 // Which object should we create?
    {
      case GVM_Tank:                                   // A new Tank instance requested
      objectList[h.second] = new Tank(*this, h.second);
      break;                                            // case GVM_Tank:
    case GVM_CommandPost:             // A new Command Post instance requested
      objectList[h.second] = new CommandPost(*this, h.second);
      break;                                     // case GVM_CommandPost:
    case GVM_Munition:                    // A new Munition instance requested
      objectList[h.second] = new Munition(*this, h.second);
      break;                                        // case GVM_Munition:
    case GVM_Ground:                        // A new Ground instance requested
      objectList[h.second] = new Ground(*this, h.second);
      break;                                          // case GVM_Ground:
    default:                                           // None of the above
      View3D::createObject(h);              // Call the parent class version
      break;                                                    // default
  }                                                          // switch (t)
}                            // void BattleView::createObject(object_handle)

void BattleView::updateTravel(void)
{                                      // void BattleView::updateTravel(void)
  up = (up<0) ? up=-speed : (up>0) ? up=speed : 0.0;
  right = (right<0) ? right=-speed : (right>0) ? right=speed : 0.0;
  forward = (forward<0) ? forward=-speed : (forward>0) ? forward=speed : 0.0;
}                                      // void BattleView::updateTravel(void)

void BattleView::begin(void)
{                                             // void BattleView::begin(void)
  pos += forward*vnv+right*vrv+up*vup;            // Get the current position
  if (pos[0]>10000.0) pos[0] = 10000.0;       // Don't go out of the play box
  if (pos[0]<-10000.0) pos[0] = -10000.0;
  if (pos[1]>10000.0) pos[1] = 10000.0;
  if (pos[1]<-10000.0) pos[1] = -10000.0;
  if (pos[2]<100.0) pos[2] = 100.0;                  // Don't go below ground
  if (zoom[0]) scale *= ZOOM_FACTOR;                   // Zoom in if desired
  if (zoom[1]) scale /= ZOOM_FACTOR;                  // Zoom out if desired
  glMatrixMode(GL_PROJECTION);                 // Establish a projection view
  glLoadIdentity();                               // Load the identity matrix
  gluPerspective(50.0, aspect, zNear, zFar);          // Establish perspecive
  glMatrixMode(GL_MODELVIEW);                         // Extablish MODELVIEW
  glLoadIdentity();                           // Load another identity matrix
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);          // How to blend
  gluLookAt(pos[0], pos[1], pos[2],
            pos[0]+vnv[0], pos[1]+vnv[1], pos[2]+vnv[2],
            vup[0], vup[1], vup[2]);    // Setup the camera viewing paramters
  glScaled(scale, scale, scale);                          // Zoom back aways
}                                             // void BattleView::begin(void)

bool BattleView::isDisplay(void)
{                                         // bool BattleView::isDisplay(void)
  return (isVisible && refresh);                      // Should we redisplay
}                                         // bool BattleView::isDisplay(void)

void BattleView::keydown(byte key, int x, int y)
{                                  // void BattleView::keydown(byte,int,int)
  glutSetWindow(window);                                   // Set the window
  switch (key)                                       // Which key was pressed
  {
    case 'W':                       // Applies to either 'w' or shift-'w'
    case 'w':
      forward = speed;                               // We're moving forward
      break;
    case 'S':                       // Applies to either 's' or shift-'s'
    case 's':
      forward = -speed;                             // We're moving backward
      break;
    case 'A':                       // Applies to either 'a' or shift-'A'
    case 'a':
      right = speed;                                    // We're moving left
      break;
    case 'D':                       // Applies to either 'a' or shift-'A'
    case 'd':
      right = -speed;                                  // We're moving right
      break;
    case 'Q':                    // Applies to either 'a' or shift-'A'
    case 'q':
      up = speed;                                      // We're moving up
      break;
    case 'E':                    // Applies to either 'e' or shift-'e'
    case 'e':
      up = -speed;                                     // We're moving up
      break;
    case '1':
    case '!': speed = 1.0; updateTravel(); break;
    case '2':
    case '@': speed = 2.0; updateTravel(); break;
    case '3':
    case '#': speed = 4.0; updateTravel(); break;
    case '4':
    case '$': speed = 8.0; updateTravel(); break;
    case '5':
    case '%': speed = 16.0; updateTravel(); break;
    case '6':
    case '^': speed = 32.0; updateTravel(); break;
    case '7':
    case '&': speed = 64.0; updateTravel(); break;
    case '8':
    case '*': speed = 128.0; updateTravel(); break;
    case '9':
    case '(': speed = 256.0; updateTravel(); break;
    case '0':
    case ')': speed = 512.0; updateTravel(); break;
    default:
      View3D::keydown(key, x, y);                   // Call parent version
  }                                                        // switch (key)
  setSceneChange(true);                         // Need to refresh display
  redisplay();                                // Redisplay the environment
}                              // void BattleView::keydown(byte,int,int)

void BattleView::keyup(byte key, int x, int y)
{                                // void BattleView::keyup(byte,int,int)
  glutSetWindow(window);                                 // Set the window
  switch (key)                                    // Which key was pressed
  {
    case 'W':                    // Applies to either 'w' or shift-'w'
    case 'w':
    case 'S':                    // Applies to either 's' or shift-'s'
    case 's':
      forward = 0.0;                      // No longermoving forward/back
      break;
    case 'A':                    // Applies to either 'a' or shift-'A'
    case 'a':
    case 'D':                    // Applies to either 'a' or shift-'A'
    case 'd':
      right = 0.0;                          // No longer moving right/left
      break;
    case 'Q':                    // Applies to either 'a' or shift-'A'
    case 'q':
    case 'E':                    // Applies to either 'e' or shift-'e'
    case 'e':
      up = 0.0;                                // No longer moving up.down
      break;
    default:
      View3D::keyup(key, x, y);                     // Call parent version
  }                                                        // switch (key)
  setSceneChange(true);                         // Need to refresh display
  redisplay();                                // Redisplay the environment
}                                // void BattleView::keyup(byte,int,int)

void BattleView::motion(int x, int y)
{                                     // void BattleView::motion(int, int)
  glutSetWindow(window);                                 // Set the window
  if (buttonState[GLUT_LEFT_BUTTON])           // If the left button is down
  {
      ori[1] += (float)(mouseLoc[1]-y)*0.125;        // Change rotation about X
      ori[2] += (float)(mouseLoc[0]-x)*0.125;        // Change rotation about Z
      if (ori[1]>90.0) ori[1]=90.0;          // Don't pitch too high or too low
      if (ori[1]<-90.0) ori[1]=-90.0;
    }                                     // if (buttonState[GLUT_LEFT_BUTTON])

    if (buttonState[GLUT_RIGHT_BUTTON])           // If the right button is down
      ori[0] += (float)(x-mouseLoc[0])*0.125;        // Change rotation about Z

    if (buttonState[GLUT_MIDDLE_BUTTON])         // If the middle button is down
    {
      float dy = ((float) (mouseLoc[1]-y))*0.2;           // Get the difference
      scale *= pow(ZOOM_FACTOR, dy);                   // Get the scaling factor
    }                                   // if (buttonState[GLUT_MIDDLE_BUTTON])

    mouseLoc[0] = x;                                      // Get the x component
    mouseLoc[1] = y;                                      // Get the y component

    GLdouble roll=ori[0]*PI/180.0;                      // Get the roll in radians
    GLdouble pitch=ori[1]*PI/180.0;                    // Get the pitch in radians
    GLdouble yaw=ori[2]*PI/180.0;                        // Get the yaw in radians

    GLdouble cx=cos(roll),  cy=cos(pitch),  cz=cos(yaw);
    GLdouble sx=sin(roll),  sy=sin(pitch),  sz=sin(yaw);

    vnv[0] = cz*cy;          vnv[1] = sz*cy;          vnv[2] = -sy;
    vrv[0] = cz*sy*sx-sz*cx; vrv[1] = sz*sy*sx+cz*cx; vrv[2] = cy*sx;
    vup[0] = cz*sy*cx+sz*sx; vup[1] = sz*sy*cx-cz*sx; vup[2] = cy*cx;

    setSceneChange(true);                              // Need to refresh display
    redisplay();                                     // Redisplay the environment
  }                                       // void BattleView::motion(int, int)
}
```

C.1.57. gvm/gvmChangeTrack.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmChangeTrack.h - Class declaration for the gvm::ChangeTrack class
////////////////////////////////////////////////////////////////////////////////

#ifndef CHANGETRACK_H_INCLUDED
#define CHANGETRACK_H_INCLUDED

#include "gvmSetNewtonianMotion.h"
#include "gvmObject.h"
#include "spt/sptNewtonianMotion.h"

#define GVM_ChangeTrack GVM_UserMessage002

namespace gvm
{                                                                // namespace gvm
  class View;                       // Forward declaration of the gvm::View class

  class ChangeTrack : public SetNewtonianMotion
  {                              // class ChangeTrack : public SetNewtonianMotion
    public:
      ChangeTrack(View&, double, object_index, const spt::NewtonianMotion&);
      ChangeTrack(View&, double, message_type, object_index,
                  const spt::NewtonianMotion&);

      virtual void send(void);                       // Deliver the message payload
  };                                        // class ChangeTrack : public Message
}                                                                // namespace gvm

#endif
```

C.1.58. gvm/gvmChangeTrack.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmChangeTrack.cxx - Class method definitions for the gvm::ChangeTrack
//                       class
```

```
////////////////////////////////////////////////////////////////////////////////

#include "gvmTacticalView.h"
#include "gvmChangeTrack.h"
#include "gvmView.h"

namespace gvm
{                                                                // namespace gvm
  ChangeTrack::ChangeTrack(View& v,
                           double t,
                           object_index i,
                           const spt::NewtonianMotion& nm)
    : SetNewtonianMotion(v, t, GVM_ChangeTrack, i, nm)
  {  // ChangeTrack::ChangeTrack(View&,double,object_index,NewtonianMotion,...)
  }  // ChangeTrack::ChangeTrack(View&,double,object_index,NewtonianMotion,...)

  ChangeTrack::ChangeTrack(View& v,
                           double t,
                           message_type ty,
                           object_index i,
                           const spt::NewtonianMotion& nm)
    : SetNewtonianMotion(v, t, ty, i, nm)
  {     // ChangeTrack::ChangeTrack(View&,double,message_type,object_index,...)
  }     // ChangeTrack::ChangeTrack(View&,double,message_type,object_index,...)

  void ChangeTrack::send(void)
  {                                             // void ChangeTrack::send(void)
    gvm::TacticalView& v = dynamic_cast<gvm::TacticalView&>(getView());
    v.changeTrack(getDest(), nm);
  }                                             // void ChangeTrack::send(void)
}                                                                // namespace gvm
```

C.1.59. gvm/gvmCommandPost.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmCommandPost.h - This draws a command post on the screen
////////////////////////////////////////////////////////////////////////////////

#ifndef GVMCOMMANDPOST_H_INCLUDED
#define GVMCOMMANDPOST_H_INCLUDED

#include "gvmNewtonianMotion.h"
#include "gvmCube.h"

#define GVM_CommandPost GVM_UserObject002

namespace gvm
{
  class View3D;

  class CommandPost : public NewtonianMotion
  {
    protected:
      Cube body;                                                // Command post

    public:
      CommandPost(gvm::View3D&, ulong);                    // Class constructor

      virtual void display(void);                   // Display the command post
      virtual bool isType(object_type);           // Check if this is of type t
  };                              // class CommandPost : public NewtonianMotion
}

#endif
```

C.1.60. gvm/gvmCommandPost.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmCommandPost.cxx - Method definitions for the CommandPost class
////////////////////////////////////////////////////////////////////////////////
```

```
#include <iostream>
#include <GL/glut.h>
#include "gvmCommandPost.h"
#include "gvmView3D.h"

namespace gvm
{
  CommandPost::CommandPost(gvm::View3D& v, ulong i)
    : NewtonianMotion(v,GVM_CommandPost,i), body(v, i)
  {                          // CommandPost::CommandPost(gvm::View3D&, ulong)
    mode = GL_LINES;                                // Set the mode to polygon
    body.set(20.0);
    body.setMode(GL_LINES);
  }                          // CommandPost::CommandPost(gvm::View3D&, ulong)


  void CommandPost::display(void)
  {                                          // void CommandPost::display(void)
    begin();
    glPushMatrix();
      glTranslated(0.0, 0.0, 10.0);                 // Put the CP above ground
      glScaled(5.0, 4.0, 1.0);                    // Main portion of the building
      body.display();                       // Display the main building portion
      glPushMatrix();
        glTranslated(0.0, 0.0, 11.0);                   // Move the roof on top
        glScaled(1.1, 1.1, 0.1);   // Make it thinner, and overhanging building
        body.display();                                     // Display the roof
      glPopMatrix();
    glPopMatrix();
    end();
  }                                          // void CommandPost::display(void)

  bool CommandPost::isType(object_type c)
  {                                      // bool CommandPost::isType(object_type)
      return (c==GVM_CommandPost || NewtonianMotion::isType(c));
  }                                      // bool CommandPost::isType(object_type)
}                                                               // namespace gvm
```

C.1.61. gvm/gvmDeleteTrack.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmDeleteTrack.h - Class declaration for the gvm::DeleteTrack class
////////////////////////////////////////////////////////////////////////////////

#ifndef DELETETRACK_H_INCLUDED
#define DELETETRACK_H_INCLUDED

#include "gvmMessage.h"
#include "gvmObject.h"

#define GVM_DeleteTrack GVM_UserMessage004

namespace gvm
{                                                               // namespace gvm
  class View;                     // Forward declaration of the gvm::View class

  class DeleteTrack : public Message
  {                                      // class DeleteTrack : public Message
    public:
      DeleteTrack(View&, double, object_index);

      virtual void send(void);                   // Deliver the message payload
  };                                     // class DeleteTrack : public Message
}                                                               // namespace gvm

#endif
```

C.1.62. gvm/gvmDeleteTrack.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmDeleteTrack.cxx - Class method definitions for the gvm::DeleteTrack
//                          class
///////////////////////////////////////////////////////////////////////////////

#include "gvmTacticalView.h"
#include "gvmDeleteTrack.h"
#include "gvmView.h"

namespace gvm
{                                                               // namespace gvm
  DeleteTrack::DeleteTrack(View& v,
                           double t,
                           object_index i)
    : Message(v, t, GVM_DeleteTrack, i)
  {                      // DeleteTrack::DeleteTrack(View&,double,object_index)
  }                      // DeleteTrack::DeleteTrack(View&,double,object_index)

  void DeleteTrack::send(void)
  {                                              // void DeleteTrack::send(void)
    gvm::TacticalView& v = dynamic_cast<gvm::TacticalView&>(getView());
    v.deleteTrack(getDest());
  }                                              // void DeleteTrack::send(void)
}                                                               // namespace gvm
```

C.1.63. gvm/gvmExplosion.h

```
///////////////////////////////////////////////////////////////////////////////
// gvmExplosion.h - Class declaration for the gvm::Explosion class
///////////////////////////////////////////////////////////////////////////////

#ifndef EXPLOSION_H_INCLUDED
#define EXPLOSION_H_INCLUDED

#include "gvmMessage.h"
#include "gvmObject.h"
#include "spt/sptDefs.h"

#define GVM_Explosion GVM_UserMessage005

namespace gvm
{                                                               // namespace gvm
  class View;                      // Forward declaration of the gvm::View class

  class Explosion : public Message
  {                                          // class Explosion : public Message
    protected:
      spt::vertex pos;                                // Location of the explosion

    public:
      Explosion(View&, double, object_index, spt::vertex);

      virtual void send(void);                     // Deliver the message payload
  };                                         // class Explosion : public Message
}                                                               // namespace gvm

#endif
```

C.1.64. gvm/gvmExplosion.cxx

```
///////////////////////////////////////////////////////////////////////////////
// gvmExplosion.cxx - Class method definitions for the gvm::Explosion class
///////////////////////////////////////////////////////////////////////////////

#include "gvmExplosion.h"
#include "gvmView.h"
#include "gvmMunition.h"

#ifdef _TRACE
#define EXPLOSION_TRACE false
#endif

namespace gvm
```

```
{                                                               // namespace gvm
  Explosion::Explosion(View& v, double t, object_index i, spt::vertex p)
    : Message(v, t, GVM_Explosion, i), pos(p)
  {              // Explosion::Explosion(View&,double,object_index,spt::vertex)
  }              // Explosion::Explosion(View&,double,object_index,spt::vertex)

  void Explosion::send(void)
  {                                                // void Explosion::send(void)
    Munition& mun = dynamic_cast<Munition&>(getView()[getDest()]);
    mun.explode(pos);                               // The munition has exploded
  }                                                // void Explosion::send(void)
}                                                               // namespace gvm
```

C.1.65. gvm/gvmGrid.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmGrid.h - This draws an m x n grid of size width x height centered at
//             (0,0)
////////////////////////////////////////////////////////////////////////////////

#ifndef GVMGRID_H_INCLUDED
#define GVMGRID_H_INCLUDED

#include "gvmShape3D.h"

#define GVM_Grid GVM_UserObject004

namespace gvm
{
  class View3D;

  class Grid : public Shape3D
  {
    protected:
      ulong m, n;            // Number of squares along x, y axis respectively
      double width, height;                     // Width and height of the grid

    public:
      Grid(View3D&,object_type,ulong,ulong,ulong,double,double);

      virtual void display(void);                  // Display the command post
      virtual bool isType(object_type);          // Check if this is of type t
  };                                   // class Grid : public NewtonianMotion
}

#endif
```

C.1.66. gvm/gvmGrid.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmGrid.cxx - Method definitions for the Grid class
////////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <GL/glut.h>
#include "gvmGrid.h"
#include "gvmView3D.h"

namespace gvm
{
  Grid::Grid(gvm::View3D& v, object_type t, ulong i,
             ulong M, ulong N, double W, double H)
    : Shape3D(v,t,i), m(M), n(N), width(W), height(H)
  {          // Grid::Grid(View3D&,object_type,ulong,ulong,ulong,double,double)
  }          // Grid::Grid(View3D&,object_type,ulong,ulong,ulong,double,double)

  void Grid::display(void)
  {                                                  // void Grid::display(void)
    double dx = width/((double) m);          // Distance between adjacet x lines
    double dy = height/((double) n);         // Distance between adjacet y lines
    double lx = width/2.0;                    // Limits of travel in x direction
    double ly = height/2.0;                   // Limits of travel in y direction

    begin();
      glPushAttrib(GL_CURRENT_BIT);                  // Get the current point size
      glColor4d(0.0, 1.0, 0.0, 0.2);           // Set color to translucent green
      glBegin(GL_LINES);
        for(double x=-lx; x<=lx; x+=dx)
        {
          glVertex2d(x, -ly);
          glVertex2d(x,  ly);
        }

        for(double y=-ly; y<=ly; y+=dy)
        {
          glVertex2d(-lx, y);
          glVertex2d( lx, y);
        }
      glEnd();
      glPopAttrib();                                         // Restore the color
    end();
  }                                                  // void Grid::display(void)

  bool Grid::isType(object_type c)
  {                                            // bool Grid::isType(object_type)
      return (c==GVM_Grid || Shape3D::isType(c));              // Return results
  }                                            // bool Grid::isType(object_type)
}                                                                // namespace gvm
```

C.1.67. gvm/gvmGround.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmGround.h - This draws the ground
////////////////////////////////////////////////////////////////////////////////

#ifndef GVMGROUND_H_INCLUDED
#define GVMGROUND_H_INCLUDED

#include "gvmGrid.h"

#define GVM_Ground GVM_UserObject005

namespace gvm
{
  class View3D;

  class Ground : public Grid
  {
    public:
      Ground(View3D&, ulong);                               // Class constructor

      virtual bool isType(object_type);           // Check if this is of type t
  };                                              // class Ground : public Grid
}

#endif
```

C.1.68. gvm/gvmGround.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmGround.cxx - Method definitions for the Ground class
////////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <GL/glut.h>
#include "gvmGround.h"
#include "gvmView.h"
```

```
namespace gvm
{
  Ground::Ground(View3D& v, ulong i)
    : Grid(v,GVM_Ground,i,20,20,20000.0,20000.0)
  {                                          // Ground::Ground(View3D&, ulong)
    mode = GL_LINES;                                  // Set the mode to polygon
  }                                          // Ground::Ground(View3D&, ulong)

  bool Ground::isType(object_type c)
  {                                        // bool Ground::isType(object_type)
      return (c==GVM_Ground || Grid::isType(c));               // Return results
  }                                      // bool Ground::isType(object_type)
}                                                             // namespace gvm
```

C.1.69. gvm/gvmMunition.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmMunition.h - This draws a tank round on the screen
////////////////////////////////////////////////////////////////////////////////

#ifndef GVMMUNITION_H_INCLUDED
#define GVMMUNITION_H_INCLUDED

#include "gvmNewtonianMotion.h"
#include "spt/sptLinearMotion.h"
#include "Random.h"

#define GVM_Munition GVM_UserObject003

namespace gvm
{
  class View3D;

  class Munition : public NewtonianMotion
  {
    protected:
      std::vector<spt::LinearMotion> fragments;          // Motion of fragments
      bool exploding;                                   // Are we exploding yet?
      sodl::Random rnd;                               // Random number generator
      double eTime;                                          // Explosion time

    public:
      Munition(gvm::View3D&, ulong);                      // Class constructor

      virtual void display(void);                   // Display the command post
      virtual bool isType(object_type);           // Check if this is of type t
      virtual void explode(spt::vertex);                 // Explode the munition

  };                               // class Munition : public NewtonianMotion
}

#endif
```

C.1.70. gvm/gvmMunition.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmMunition.cxx - Method definitions for the Munition class
////////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <GL/glut.h>
#include "gvmMunition.h"
#include "gvmView3D.h"

#define FRAGMENT_COUNT 500

namespace gvm
{
  Munition::Munition(gvm::View3D& v, ulong i)
    : NewtonianMotion(v,GVM_Munition,i), fragments(FRAGMENT_COUNT),
      exploding(false), eTime(0.0)
```

```
{                                    // Munition::Munition(gvm::View3D&, ulong)
  mode = GL_LINES;                                 // Set the mode to polygon
}                                    // Munition::Munition(gvm::View3D&, ulong)

void Munition::display(void)
{                                               // void Munition::display(void)
  glPushAttrib(GL_POINT_BIT);                     // Save the current point size
  glPushAttrib(GL_CURRENT_BIT);                // Save the current drawing color
  glPointSize(3.0);                                       // Set the point size

  glColor3d(1.0, 1.0, 1.0);       // Set the color of the projectile to white

  if (!exploding)            // If we are on the initial flight to the target
  {
    glColor3d(1.0, 1.0, 1.0);     // Set the color of the projectile to white
    begin();
      glBegin(GL_POINTS);                                  // Want to do points
        glVertex3d(0.0, 0.0, 0.0);                // Well, one of them any way
      glEnd();                                                     // That's it
    end();                                                     // All done here
  }                                                          // if (!exploding)
  else                                       // If we're doing the explosion now
  {
    double t=getView().getTime();                       // Get the current time
    double dt = t-eTime;      // Difference between current & explosion times
    if (dt >= 5.0) setActive(false);          // Turn off after 5 sim-seconds
    double alpha = 1.0-(dt/5.0);      // Specify alpha component of fragments
    glEnable(GL_BLEND);                               // Enable alpha blending
    glColor4d(1.0, 1.0, 1.0, alpha);                           // Set new color
    glBegin(GL_POINTS);                                   // Draw the particles
      for (ulong i=0; i<fragments.size(); ++i)     // Loop over all fragments
      {
        while (t>fragments[i].getStopTime())       // If we're doing a bounce
        {
          double stop = fragments[i].getStopTime();  // Get current stop time
          spt::vertex p = fragments[i].lp(stop); // Get position at stop time
         spt::vertex v = fragments[i].lv(stop); // Get velocity at stop time
          spt::vertex a = fragments[i].la(stop);      // Get acc at stop time
          v *= 0.9;                                              //Energy loss
          v[2] = -v[2];                                               // Bounce
          double newStop = stop-2.0*v[2]/a[2];        // Get next bounce time
          fragments[i].setLM(p, v, a, stop, newStop);          // Set next leg
        }                               // if (t>fragments[i].getStopTime())
        spt::vertex pos = fragments[i].lp(t);     // Get fragment current pos
        glVertex3d(pos[0], pos[1], pos[2]);                // Draw the fragment
      }                           // for (ulong i=0; i<fragments.size(); ++i)
    glEnd();                                          // glBegin(GL_POLYGON)
    glDisable(GL_BLEND);                             // Disable alpha blending
  }                                                // else from if (!exploding)
  glPopAttrib();                                           // Restore the color
  glPopAttrib();                                // Restore the point attributes
}                                               // void Munition::display(void)

void Munition::explode(spt::vertex p)
{                                        // void Munition::explode(spt::vertex)
  spt::vertex dv(0.0, 3);               // Delta from the main velocity vector
  spt::vertex a(0.0, 3);                                        // Acceleration
  a[2] = -9.8;                                                // Due to gravity

  exploding = true;                                   // We are now exploding;
  eTime = getView().getTime();                      // Save the explosion time
  for (ulong i=0; i<fragments.size(); ++i)  // Loop over all of the fragments
  {
    double theta = rnd.nextDouble(0, 2.0*PI);
    double phi = rnd.nextDouble(0.0, PI);
    double rad = rnd.nextDouble(0, 20.0);
    dv[0] = rad*cos(phi)*cos(theta);
    dv[1] = rad*cos(phi)*sin(theta);
    dv[2] = rad*sin(phi)+5.0;
    double stop = eTime-2.0*(dv[2])/a[2];
    fragments[i].setLM(p, dv, a, eTime, stop);
    }                                // for (ulong i=0; i<fragments.size(); ++i)
  }                                       // void Munition::explode(spt::vertex)

  bool Munition::isType(object_type c)
  {                                        // bool Munition::isType(object_type)
    return (c==GVM_Munition || NewtonianMotion::isType(c)); // Return results
  }                                        // bool Munition::isType(object_type)
}                                                              // namespace gvm
```

C.1.71. gvm/gvmNewtonianMotion.h

```
///////////////////////////////////////////////////////////////////////////////
// gvmNewtonianMotion.h - This draws a command post on the screen
///////////////////////////////////////////////////////////////////////////////

#ifndef GVMNEWTONIANMOTION_H_INCLUDED
#define GVMNEWTONIANMOTION_H_INCLUDED

#include "gvmShape3D.h"
#include "spt/sptNewtonianMotion.h"

#define GVM_NewtonianMotion GVM_UserObject000

namespace gvm
{
  class View3D;

  class NewtonianMotion : public Shape3D
  {
    protected:
      NewtonianMotion(gvm::View3D&, object_type, ulong);   // Class constructor
      spt::NewtonianMotion nm;                      // Newtonian motion parameters

    public:
      virtual void setNM(const spt::NewtonianMotion&);     // Update parameters
      virtual void begin(void);                     // Begin displaying the object
      virtual void end(void);             // We're all finished with the object
      virtual bool isType(object_type);           // Check if this is of type t
  };                                     // class NewtonianMotion : public Shape3D
}

#endif
```

C.1.72. gvm/gvmNewtonianMotion.cxx

```
///////////////////////////////////////////////////////////////////////////////
// gvmNewtonianMotion.cxx - Method definitions for the NewtonianMotion class
///////////////////////////////////////////////////////////////////////////////

#include <GL/glut.h>
#include "gvmNewtonianMotion.h"
#include "gvmView3D.h"
#include <math.h>
#include "spt/sptDefs.h"

namespace gvm
{
  NewtonianMotion::NewtonianMotion(gvm::View3D& v, object_type t, ulong i)
    : Shape3D(v, t, i)
  {                    // NewtonianMotion::NewtonianMotion(gvm::View3D&, ulong)
  }                    // NewtonianMotion::NewtonianMotion(gvm::View3D&, ulong)

  void NewtonianMotion::begin(void)
  {                                         // void NewtonianMotion::begin(void)
    double time = getView().getTime();                         // Get current time
    spt::vertex pos = nm.lp(time);                        // Get current position
    spt::vertex rot = 180.0*nm.ap(time)/PI;                    // Get orientation

    glPushMatrix();                                    // Push the current matrix
      glTranslated(pos[0], pos[1], pos[2]);       // Move to the proper position
      glRotated(rot[0], 1.0, 0.0, 0.0);                                   // Roll
      glRotated(rot[1], 0.0, 1.0, 0.0);                               // Pitch
      glRotated(rot[2], 0.0, 0.0, 1.0);                                 // Yaw
      Shape3D::begin();          // Call the parent version of the begin method
    }                                      // void NewtonianMotion::begin(void)

    void NewtonianMotion::end(void)
    {                                        // void NewtonianMotion::end(void)
      Shape3D::end();              // Call the parent version of the end method
    glPopMatrix();                                            // Pop the matrix
    }                                        // void NewtonianMotion::end(void)

    void NewtonianMotion::setNM(const spt::NewtonianMotion& n)
    {                // void NewtonianMotion::setNM(const spt::NewtonianMotion&)
    nm=n;                                  // Set the newtonin motion paramters
    }                // void NewtonianMotion::setNM(const spt::NewtonianMotion&)

    bool NewtonianMotion::isType(object_type c)
    {                              // bool NewtonianMotion::isType(object_type)
      return (c==GVM_NewtonianMotion || Shape3D::isType(c));  // Return results
    }                              // bool NewtonianMotion::isType(object_type)
  }                                                            // namespace gvm
```

C.1.73. gvm/gvmSetNewtonianMotion.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmSetNewtonianMotion.h - Class declaration for the
//                                gvm::SetNewtonianMotion class
////////////////////////////////////////////////////////////////////////////////

#ifndef SETNEWTONIANPOSITION_H_INCLUDED
#define SETNEWTONIANPOSITION_H_INCLUDED

#include "gvmMessage.h"
#include "gvmObject.h"
#include "spt/sptNewtonianMotion.h"

#define GVM_SetNewtonianMotion GVM_UserMessage001

namespace gvm
{                                                              // namespace gvm
  class View;                   // Forward declaration of the gvm::View class

  class SetNewtonianMotion : public Message
  {                              // class SetNewtonianMotion : public Message
    protected:
      spt::NewtonianMotion nm;                      // Netonian motion parameters

    public:
      SetNewtonianMotion(View&, double, object_index,
                         const spt::NewtonianMotion&);
      SetNewtonianMotion(View&, double, message_type, object_index,
                         const spt::NewtonianMotion&);

      virtual void send(void);                    // Deliver the message payload
  };                            // class SetNewtonianMotion : public SetMotion
}                                                              // namespace gvm

#endif
```

C.1.74. gvm/gvmSetNewtonianMotion.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmSetNewtonianMotion.cxx - Class method definitions for the
//                                  gvm::SetNewtonianMotion class
////////////////////////////////////////////////////////////////////////////////

#include "gvmSetNewtonianMotion.h"
#include "gvmView.h"
#include "gvmNewtonianMotion.h"

namespace gvm
```

```
{                                                               // namespace gvm
  SetNewtonianMotion::SetNewtonianMotion(View& v,
                                         double t,
                                         object_index i,
                                         const spt::NewtonianMotion& n)
    : Message(v, t, GVM_SetNewtonianMotion, i), nm(n)
  {         // SetNewtonianMotion::SetNewtonianMotion(View&, double, ... )
  }         // SetNewtonianMotion::SetNewtonianMotion(View&, double, ... )

  SetNewtonianMotion::SetNewtonianMotion(View& v,
                                         double t,
                                         message_type ty,
                                         object_index i,
                                         const spt::NewtonianMotion& n)
    : Message(v, t, ty, i), nm(n)
  {         // SetNewtonianMotion::SetNewtonianMotion(View&, double, ... )
  }         // SetNewtonianMotion::SetNewtonianMotion(View&, double, ... )

  void SetNewtonianMotion::send(void)
  {                                     // void SetNewtonianMotion::send(void)
    NewtonianMotion* m=dynamic_cast<NewtonianMotion*>(&getView()[getDest()]);
    if (m!=NULL) m->setNM(nm);   // Set destination Newtonian motion parameters
  }                                     // void SetNewtonianMotion::send(void)
}                                                               // namespace gvm
```

C.1.75. gvm/gvmSetTankState.h

```
//////////////////////////////////////////////////////////////////////////////
// gvmSetTankState.h - Class declaration for the gvm::SetTankState class
//////////////////////////////////////////////////////////////////////////////

#ifndef SETTANKSTATE_H_INCLUDED
#define SETTANKSTATE_H_INCLUDED

#include "gvmMessage.h"
#include "gvmObject.h"

#define GVM_SetTankState GVM_UserMessage000

namespace gvm
{                                                               // namespace gvm
  class View;                   // Forward declaration of the gvm::View class

  class SetTankState : public Message
  {                                   // class SetTankState : public Message
    private:
      double az;                                                // Gun azimuth
      double azRate;                                      // Azimuth slew rate
      double azStart;                               // Azimuth slew start time
      double azStop;                                // Azimuth slew start time
      double el;                                              // Gun elevation
      double elRate;                                    // Elevation slew rate
      double elStart;                             // Elevation slew start time
      double elStop;                              // Elevation slew start time

    public:
      SetTankState(View&, double, object_index, double, double, double, double,
                   double, double, double, double);

      virtual void send(void);                   // Deliver the message payload
  };                                   // class SetTankState : public Message
}                                                               // namespace gvm
#endif
```

C.1.76. gvm/gvmSetTankState.cxx

```
//////////////////////////////////////////////////////////////////////////////
// gvmSetTankState.cxx - Class method definitions for the gvm::SetTankState
//                       class
//////////////////////////////////////////////////////////////////////////////
```

```
#include "gvmSetTankState.h"
#include "gvmView.h"
#include "gvmTank.h"

#ifdef _TRACE
#define SETTANKSTATE_TRACE false
#endif

namespace gvm
{                                                                    // namespace gvm
SetTankState::SetTankState(View& v,
                             double t,
                             object_index i,
                             double a,
                             double ar,
                             double as0,
                             double as1,
                             double e,
                             double er,
                             double es0,
                             double es1)

   : Message(v, t, GVM_SetTankState, i), az(a), azRate(ar), azStart(as0),
     azStop(as1), el(e), elRate(er), elStart(es0), elStop(es1)
  {  // SetTankState::SetTankState(View&, double, object_index, GLdouble, ... )
  }  // SetTankState::SetTankState(View&, double, object_index, GLdouble, ... )

  void SetTankState::send(void)
  {                                                 // void SetTankState::send(void)
    gvm::Tank &t = dynamic_cast<gvm::Tank&>(getView()[getDest()]);
    t.setTank(az, azRate, azStart, azStop, el, elRate, elStart, elStop);
  }                                                 // void SetTankState::send(void)
}                                                                    // namespace gvm
```

C.1.77. gvm/gvmTacticalGrid.h

```
//////////////////////////////////////////////////////////////////////////////
// gvmTacticalGrid.h - This draws the ground
//////////////////////////////////////////////////////////////////////////////

#ifndef GVMTACTICALGRID_H_INCLUDED
#define GVMTACTICALGRID_H_INCLUDED

#include "gvmGrid.h"

#define GVM_TacticalGrid GVM_UserObject006

namespace gvm
{
  class View3D;

  class TacticalGrid : public Grid
  {
    public:
      TacticalGrid(View3D&, ulong);                         // Class constructor

      virtual bool isType(object_type);           // Check if this is of type t
  };                                          // class TacticalGrid : public Grid
}

#endif
```

C.1.78. gvm/gvmTacticalGrid.cxx

```
//////////////////////////////////////////////////////////////////////////////
// gvmTacticalGrid.cxx - Method definitions for the TacticalGrid class
//////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <GL/glut.h>
#include "gvmTacticalGrid.h"
```

```
#include "gvmView.h"

namespace gvm
{
  TacticalGrid::TacticalGrid(View3D& v, ulong i)
    : Grid(v,GVM_TacticalGrid,i,20,20,20000.0,20000.0)
  {                                // TacticalGrid::TacticalGrid(View3D&, ulong)
    mode = GL_LINES;                                  // Set the mode to polygon
  }                                // TacticalGrid::TacticalGrid(View3D&, ulong)

  bool TacticalGrid::isType(object_type c)
  {                                    // bool TacticalGrid::isType(object_type)
      return (c==GVM_TacticalGrid || Grid::isType(c));        // Return results
  }                                    // bool TacticalGrid::isType(object_type)
}                                                              // namespace gvm
```

C.1.79. gvm/gvmTacticalView.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmTacticalView.h - Defines the gvm::TacticalView class in which all
//                     gvm::Object instances are viewed.
////////////////////////////////////////////////////////////////////////////////

#ifndef GVMTACTICALVIEW_H_INCLUDED
#define GVMTACTICALVIEW_H_INCLUDED

#include "spt/sptEnvironmentObject.h"
#include "gvmView3D.h"
#include "gvmTrack.h"
#include <vector>

namespace gvm
{                                                              // namespace gvm
  class TacticalGrid;

  class TacticalView : public View3D
  {                                        // class TacticalView : public View3D
    protected:
      GLint width, height;        // Width and height of the tactical view port
      TacticalGrid *grid;                                      // Refrence grid
      std::vector<Track> trackList;                       // List of the tracks

    public:
      TacticalView(void);                                  // Class constructor

      virtual void reshape(int, int);            // Callback for window resizing

      virtual void createObject(object_handle);     // Schedule object creation
      virtual void begin(void);              // Start rendering the scene graph
      virtual void display(void);                         // Display the tracks
      virtual void end(void);                 // Stop rendering the scene graph
      virtual bool isDisplay(void);          // Should we update the display?
      virtual void addTrack(object_index, spt::NewtonianMotion, double, ulong);
      virtual void changeTrack(object_index, spt::NewtonianMotion);
      virtual void deleteTrack(object_index);
  };                                       // class TacticalView : public View3D
}                                                              // namespace gvm

#endif
```

C.1.80. gvm/gvmTacticalView.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmTacticalView.cxx - Class member and static definitions of the
//                       gvm::TacticalView class.
////////////////////////////////////////////////////////////////////////////////

#ifdef _TRACE
#define TACTICALVIEW_TRACE false
#endif
```

```
#include "Exception.h"
#include "gvmTacticalView.h"
#include "gvmTacticalGrid.h"
#include "EngineStand.h"

namespace gvm
{                                                                   // namespace gvm
  TacticalView::TacticalView(void)                            // Window for the view
  {                                                   // TacticalView::TacticalView(void)
    setSceneChange(true);                                    // Need to refresh display
    redisplay();                                           // Redisplay the environment
    GLint vp[4];                                                // Viewport parameters
    glGetIntegerv(GL_VIEWPORT, vp);             // Get the window viewport params
    width = vp[2]; height = vp[3];  // Get the height and width of the viewport
    grid = new TacticalGrid(*this, 0);                                   // Add the grid
    grid->setColor(0.0, 0.0, 1.0, 1.0);                           // Set the grid color

    for (ulong i=0; i<100; ++i)                 // Allocate an initial 100 tracks
    {
      trackList.push_back(gvm::Track(*this,i,-1.0,NEUTRAL));
      trackList.back().setActive(false);                       // Set the active flag
    }                                                // while (trackList.size()<=i)
  }                                                 // TacticalView::TacticalView(void)

  void TacticalView::reshape(int w, int h)
  {                                          // void TacticalView::reshape(int, int)
    View3D::reshape(w,h);
    width = w; height = h;
    redisplay();
  }                                          // void TacticalView::reshape(int, int)

  void TacticalView::createObject(object_handle h)
  {                               // void TacticalView::createObject(object_handle)
  }                               // void TacticalView::createObject(object_handle)

  void TacticalView::begin(void)
  {                                                  // void TacticalView::begin(void)
    glutSetWindow(window);                                          // Set the window
    glMatrixMode(GL_PROJECTION);                      // Establish a projection view
    glLoadIdentity();                                    // Load the identity matrix
    gluPerspective(90.0, aspect, 9000.0, 11000.0);        // Establish perspecive
    glMatrixMode(GL_MODELVIEW);                                // Extablish MODELVIEW
    glLoadIdentity();                                // Load another identity matrix
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);               // How to blend
    glRotated(-90.0, 0.0, 0.0, 1.0);                     // Get it corectly positioned
    glRotated(180.0, 0.0, 1.0, 0.0);                     // Get it corectly positioned
    glTranslated(0.0, 0.0, 10500.0);                    // Position the view
  }                                                  // void TacticalView::begin(void)

  void TacticalView::display(void)
  {                                                // void TacticalView::display(void)
    if (isDisplay() || sodl::EngineStand::stand.holding()) // Update the scene?
    {
      begin();                            // Let derived classes do what they need
      GLboolean smooth = glIsEnabled(GL_POINT_SMOOTH);     // Setting to restore
      if (!smooth) glEnable(GL_POINT_SMOOTH);            // Create smooth points
      glPointSize(ptSize);                                     // Set the point size

      for (ulong i=0; i<trackList.size(); ++i) trackList[i].displaySensor();
      grid->display();                                               // Display the grid
      for (ulong i=0; i<trackList.size(); ++i) trackList[i].displayTrack();

      setRefresh(sodl::EngineStand::stand.holding()); // Set the refresh value
      setSceneChange(false);                            // The scene has not changed
      if (!smooth) glDisable(GL_POINT_SMOOTH);           // Create smooth points
      end();                               // Let derived classes do the cleanup
    }                                                                // if (isDisplay)
  }                                                // void TacticalView::display(void)

  void TacticalView::end(void)
  {                                                    // void TacticalView::end(void)
```

```
    glutSwapBuffers();                                      // Swap the buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);     // Reset the buffer
  }                                                  // void TacticalView::end(void)

  void TacticalView::addTrack(object_index i,                         // Track ID #
                              spt::NewtonianMotion nm,      // Motion parameters
                              double r,                         // Sensor radius
                              ulong f)                        // Force indicator
  {      // void TacticalView::addTrack(object_index,spt::NewtonianMotion,...)
    trackList[i].setTrack(r, f);                    // Set the track parameters
    trackList[i].setNM(nm);                        // Set the motion parameters
    trackList[i].setActive(true);                           // Turn the track on
  }      // void TacticalView::addTrack(object_index,spt::NewtonianMotion,...)

  void TacticalView::changeTrack(object_index i,                      // Track ID #
                                 spt::NewtonianMotion nm)  // Motion parameters
  {       // void TacticalView::changeTrack(object_index,spt::NewtonianMotion)
    if (trackList[i].isActive()) trackList[i].setNM(nm); // If it's active, set
  }       // void TacticalView::changeTrack(object_index,spt::NewtonianMotion)

  void TacticalView::deleteTrack(object_index i)                     // Track ID #
  {                              // void TacticalView::deleteTrack(object_index)
    trackList[i].setActive(false);                  // Set the track parameters
  }                              // void TacticalView::deleteTrack(object_index)

  bool TacticalView::isDisplay(void)
  {                                        // bool TacticalView::isDisplay(void)
    return (isVisible && refresh);                       // Should we redisplay
  }                                        // bool TacticalView::isDisplay(void)
}
```

C.1.81. gvm/gvmTank.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmTank.h - This draws a tank on the screen
////////////////////////////////////////////////////////////////////////////////

#ifndef GVMTANK_H_INCLUDED
#define GVMTANK_H_INCLUDED

#include "gvmNewtonianMotion.h"
#include "gvmCube.h"
#include "gvmCylinder.h"
#include "gvmSphere.h"

#define GVM_Tank GVM_UserObject001
#define LINE_ABREAST 0
#define V_FORMATION 1
#define FORWARD_SWEEP 2
#define COLUMN 3

namespace gvm
{
  class View3D;

  class Tank : public NewtonianMotion
  {
    protected:
      double az;                                                // Gun azimuth
      double azRate;                                      // Azimuth slew rate
      double azStart;                               // Azimuth slew start time
      double azStop;                                // Azimuth slew start time
      double el;                                              // Gun elevation
      double elRate;                                    // Elevation slew rate
      double elStart;                             // Elevation slew start time
      double elStop;                              // Elevation slew start time

      Cylinder gun;                                                // Tank gun
      Sphere turret;                                            // Tank turret
      Cube body;                                                  // Tank body
    public:
      Tank(gvm::View3D&, ulong);                              // Class constructor

      virtual double getAzimuth(double);             // Get the current azimuth
      virtual double getElevation(double);         // Get the current elevation
      virtual void display(void);                              // Display the tank
      virtual void setTank(double,double,double,double,double,double,double,
                     double);
      virtual bool isType(object_type);           // Check if this is of type t
  };                                         // class Tank : public NewtonianMotion
}

#endif
```

C.1.82. gvm/gvmTank.cxx

```
//////////////////////////////////////////////////////////////////////////////
// gvmTank.cxx - Method definitions for the Tank class
//////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <GL/glut.h>
#include "gvmTank.h"
#include "gvmView3D.h"

double deg(double r) { return 180.0*r/PI; }
double rad(double d) { return PI*d/180.0; }

namespace gvm
{
  Tank::Tank(gvm::View3D& v, ulong i)
    : NewtonianMotion(v,GVM_Tank,i), az(0.0), azRate(0.0), azStart(0.0),
      azStop(0.0), el(0.0), elRate(0.0), elStart(0.0), elStop(0.0), gun(v,i),
      turret(v,i), body(v,i)
  {                                          // Tank::Tank(gvm::View3D&, ulong)
    mode = GL_LINES;                                  // Set the mode to polygon
    gun.set(0.5, 15.0, 10, 10);
    gun.setMode(GL_LINES);
    turret.set(1.5, 10, 10);
    turret.setMode(GL_LINES);
    body.set(1.0);
    body.setMode(GL_LINES);
  }                                          // Tank::Tank(gvm::View3D&, ulong)

  void Tank::display(void)
  {                                                   // void Tank::display(void)
    double t = getView().getTime();    // Get current time for display purposes

    begin();                                                 // Perform the setup
    glPushMatrix();
      glTranslated(0.0, 0.0, 1.0);                  // Move tank above the ground
      glPushMatrix();
        glScaled(15.0, 10, 1.0);          // Outer, thinner portion of tank body
        body.display();                                       // Display the body
      glPopMatrix();

      glPushMatrix();
        glScaled(10.0, 7.5, 2.0);      // Inner, fatter portion of the tank body
        body.display();                                      // Display that part
      glPopMatrix();

      glPushMatrix();
        glTranslated(2.0, 0.0, 2.25);   // Move turret assy forward from center
        glRotated(deg(getAzimuth(t)),0.0,0.0,1.0);          // Set turret azimuth
        glRotated(90.0-deg(getElevation(t)),0.0,1.0,0.0);  // Set gun elevation
        turret.display();                            // Display the turret portion
        glPushMatrix();
          glTranslated(0.0, 0.0, 7.5);            // Change the center of the gun
          gun.display();                                       // Display the gun
          glPushMatrix();
            glTranslated(0.0, 0.0, 7.5);     // Final embelishment on end of gun
            glScaled(1.0, 1.5, 0.05);                 // Slightly wider than high
            gun.display();                              // Redisplay the cylinder
            glPopMatrix();
          glPopMatrix();
        glPopMatrix();
      glPopMatrix();
      end();
    }                                                 // void Tank::display(void)

    double Tank::getAzimuth(double t)                  // Effective time of azimuth
    {                                            // double Tank::getAzimuth(double)
      double dt = min(azStop, t)-azStart;                   // Get the elapsed time
      double rv = fmod(az+azRate*dt, 2.0*PI);      // Get the raw azimuth position
      return (rv < 0.0 ? rv+2.0*PI : rv);               // Return in range [0, 2*PI)
    }                                            // double Tank::getAzimuth(double)

    double Tank::getElevation(double t)              // Effective time of elevation
    {                                          // double Tank::getElevation(double)
      double dt = min(elStop, t)-elStart;                   // Get the elapsed time
      double rv = fmod(el+elRate*dt, 2.0*PI);    // Get the raw elevation position
      return (rv < 0.0 ? rv+2.0*PI : rv);               // Return in range [0, 2*PI)
    }                                          // double Tank::getElevation(double)

    void Tank::setTank(double a, double ar, double as0, double as1,
                       double e, double er, double es0, double es1)
    {     // void Tank::setTanl(double,double,double,double,double,double, ... )
      az = a; azRate = ar; azStart = as0; azStop = as1;
      el = e; elRate = er; elStart = es0; elStop = es1;
      getView().setSceneChange(true);               // Don't need to refresh again
    }     // void Tank::setTanl(double,double,double,double,double,double, ... )

    bool Tank::isType(object_type c)
    {                                            // bool Tank::isType(object_type)
        return (c==GVM_Tank || NewtonianMotion::isType(c));     // Return results
    }                                            // bool Tank::isType(object_type)
}                                                                // namespace gvm
```

C.1.83. gvm/gvmTrack.h

```
////////////////////////////////////////////////////////////////////////////////
// gvmTrack.h - This draws a tank on the screen
////////////////////////////////////////////////////////////////////////////////

#ifndef GVMTRACK_H_INCLUDED
#define GVMTRACK_H_INCLUDED

#include "gvmNewtonianMotion.h"

#define GVM_Track GVM_UserObject007

namespace gvm
{
  class View3D;

  class Track : public NewtonianMotion
  {
    protected:
      double radius;                                             // Sensor radius
      ulong force;                                            // Force indicator
      static std::vector<std::pair<double, double> > disc;      // Disc elements

    public:
      Track(gvm::View3D&, ulong, double, ulong);             // Class constructor

      virtual void displayTrack(void);                       // Display the track
      virtual void displaySensor(void);     // Display sensor range information
      virtual void setTrack(double, ulong);          // Set the track parameters
      virtual bool isType(object_type);           // Check if this is of type t
  };                                      // class Track : public NewtonianMotion
}
```

```
#endif
```

C.1.84. gvm/gvmTrack.cxx

```
////////////////////////////////////////////////////////////////////////////////
// gvmTrack.cxx - Method definitions for the Track class
////////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <GL/glut.h>
#include "gvmTrack.h"
#include "gvmView3D.h"
#include "spt/sptEnvironmentObject.h"

#define DISC_SIZE 64

namespace gvm
{
  std::vector<std::pair<double, double> > Track::disc;

  Track::Track(gvm::View3D& v, ulong i, double r, ulong f)
    : NewtonianMotion(v,GVM_Track,i), radius(r), force(f)
  {                                         // Track::Track(gvm::View3D&, ulong)
    mode = GL_LINES;                                 // Set the mode to polygon
    if (disc.empty())    // If the disc components have not yet been initialized
    {
      double dt = 2.0*PI/((double) DISC_SIZE);                    // Delta theta
      for (double theta=0; theta<2.0*PI; theta+=dt)            // loop over disc
        disc.push_back(make_pair(cos(theta), sin(theta)));     // Get each point
    }                                                        // if (disc.empty())

    if (force == BLUE) setColor(0.0, 0.0, 1.0, 1.0);
    else if (force == RED) setColor(1.0, 0.0, 0.0, 1.0);
    else setColor(1.0, 1.0, 1.0, 1.0);
  }                                         // Track::Track(gvm::View3D&, ulong)

  void Track::displayTrack(void)
  {                                            // void Track::displayTrack(void)
    if (isActive())                                // If this is an active track
    {
      begin();                                              // Perform the setup
      glBegin(GL_QUADS);
        glVertex2d(200.0, 0.0);
        glVertex2d(0.0, -50.0);
        glVertex2d(-50.0, 0.0);
        glVertex2d(0.0,  50.0);
      glEnd();
      end();                                      // Stop displaying this track
    }                                                         // if (isActive())
  }                                            // void Track::displayTrack(void)

  void Track::displaySensor(void)
  {                                           // void Track::displaySensor(void)
    if (isActive() && radius>0.0)                  // If this is an active track
    {
      ulong i;                                       // For loop index variable
      GLdouble dred = (force == BLUE ? 0.0 : 0.5);
      GLdouble dgreen = (force == RED || force == BLUE ? 0.0 : 0.5);
      GLdouble dblue = (force == RED ? 0.0 : 0.5);

      begin();                                              // Perform the setup
        glEnable(GL_BLEND);                             // Enable alpha blending
        glColor4d(dred, dgreen, dblue, 0.4);                    // Set new color
        glBegin(GL_POLYGON);                          // Draw the filled portion
        for(i=0; i<disc.size(); ++i)         // Loop over the points in the disc
          glVertex2d(radius*disc[i].first, radius*disc[i].second);
        glEnd();                                          // glBegin(GL_POLYGON)
        glDisable(GL_BLEND);                           // Disable alpha blending
      end();                                      // Stop displaying this track
    }                                                         // if (isActive())
  }                                           // void Track::displaySensor(void)
  void Track::setTrack(double r, ulong f)
  {                                       // void Track::setTrack(double, ulong)
    radius = r;
    force = f;

    if (force == BLUE) setColor(0.0, 0.0, 1.0, 1.0);
    else if (force == RED) setColor(1.0, 0.0, 0.0, 1.0);
    else setColor(1.0, 1.0, 1.0, 1.0);
  }                                       // void Track::setTrack(double, ulong)

  bool Track::isType(object_type c)
  {                                           // bool Track::isType(object_type)
      return (c==GVM_Track || NewtonianMotion::isType(c));    // Return results
  }                                           // bool Track::isType(object_type)
}                                                               // namespace gvm
```

C.1.85. spt/sptAngularMotion.h

```
{import process {NewtonianMotion} }
{import message {AddTrack, ChangeTrack, LoseTrack, AddEnvironment,
                 SetEnvironment, SetNewtonianMotion, Impact, Destroyed} }
{import spt {sptEnvironmentObject, sptNewtonianMotion} }
{import std {<vector>} }

{
  process:SensorTrack(NewtonianMotion)
  {                                       // process:SensorTrack(NewtonianMotion)
    ulong:envIndex((ulong) (-1));              // Index within the environment
    process:environment;                                // Environment process
    process:parent;                         // Parent object to report back to
    double:radius(2000.0);                                   // Sensor Radius
    ulong:force(NEUTRAL);          // Initially neutral, until we know better
    spt::NewtonianMotion:tracks[];               // Collection of known tracks
    std::set<ulong>:active;                     // Collection of active tracks

    method:isActive(public; bool; ulong:i;)               // Is track i active?
    { return active.find(i)!=active.end(); }

    method:notify(public; void; std::vector<SetNewtonianMotion>&:out;)
    {     // method:notify(public; void; std::vector<SetNewtonianMotion>&:out;)
      NewtonianMotion::notify(out);                    // Call the parent version
      if (envIndex != ((ulong) -1))   // If we have registered with environment
      {
        out.push_back(me);                           // Allocate a new message
        out.back().addDest(environment);            // Add environment as dest
        out.back().addDest(parent);                 // Add environment as dest
        out.back().index = envIndex;                      // Specify the index
        out.back().set(nm);              // Specify the Linear Motion paramters
      }                                         // if (envIndex != ((ulong) -1))
    }     // method:notify(public; void; std::vector<SetNewtonianMotion>&:out;)

    mode:Default
    {                                                           // mode:Default
      node:addEnvironment[AddEnvironment:in]
                        [SetNewtonianMotion:out=>(parent;)]
      {                   // Node:addEnvironment[AddEnvironment:in][...]
        envIndex = in.index;                     // Save the environment index
        out.index = envIndex;                    // Notify parent of new index
        out.set(nm);                              // Set the motion parameters
      }                   // Node:addEnvironment[AddEnvironment:in][...]

      node:setEnvironment[SetEnvironment:in][]
      {                          // node:setEnvironment[SetEnvironment:in][]
        environment = in.environment;       // Environment in which this exists
        parent = in.getSource();             // Save the parent process handle
      }                          // node:setEnvironment[SetEnvironment:in][]

      node:addTrack[AddTrack:in][AddTrack:out=>(parent;)]
      {                          // node:addTrack[AddTrack:in][AddTrack:out]
        if (tracks.size() <= in.getTrack())       // If tracks isn't big enough
```

```
          tracks.resize(in.getTrack()+1);              // Resize the track list

        tracks[in.getTrack()] = in.getMotion();      // Save the motion paramter
        active.insert(in.getTrack());                   // Add an active element
        out.setMotion(in.getMotion());              // Notify as to object motion
        out.set(in.getTrack(), in.getForce());         // Provide force tracking
      }                              // node:addTrack[AddTrack:in][AddTrack:out]

      node:changeTrack[ChangeTrack:in][ChangeTrack:out=>(parent;)]
      {                     // node:changeTrack[ChangeTrack:in][ChangeTrack:out]
        tracks[in.getTrack()] = in.getMotion();      // Save the motion paramter
        out.setMotion(in.getMotion());              // Notify as to object motion
        out.set(in.getTrack(), in.getForce());         // Provide force tracking
      }                     // node:changeTrack[ChangeTrack:in][ChangeTrack:out]

      node:loseTrack[LoseTrack:in][LoseTrack:out=>(parent;)]
      {                           // node:loseTrack[LoseTrack:in][LoseTrack:out]
        active.erase(in.getTrack());                // Track is no longer active
        out.set(in.getTrack(),in.getForce());      // Provide force information
      }                           // node:loseTrack[LoseTrack:in][LoseTrack:out]

      node:impact[Impact:in]                                  // We've been hit
                 [Destroyed:out[],       // Notify processes of our destruction
                  LoseTrack:lt[]=>(parent;),             // Lost all the tracks
                  SetNewtonianMotion:snm[]]   // Notify of new newtonian motion
      {                              // node:impact[Impact:in][Destroyed:out[]]
        nm.la(0.0, 0.0, 0.0, getTime());           // Stop linear acceleration
        nm.lv(0.0, 0.0, 0.0, getTime());                 // Stop linear motion
        nm.aa(0.0, 0.0, 0.0, getTime());          // Stop angular acceleration
        nm.av(0.0, 0.0, 0.0, getTime());                // Stop angular motion
        notify(snm);     // Notify views/environments about new newtonian motion

        std::map<process, gvm::object_index>::iterator i;     // For loop index
        for (i=views.begin(); i!=views.end(); ++i)       // Loop over index map
        {
          out.push_back(me);                            // Allocate a new message
          out.back().addDest(i->first);       // Add this view as a destination
          out.back().index = i->second;                     // Specify the index
        }                          // for (i=views.begin(); i!=views.end(); ++i)

        if (envIndex != ((ulong) -1)) // If we have registered with environment
        {
          out.push_back(me);                            // Allocate a new message
          out.back().addDest(environment);            // Add environment as dest
          out.back().addDest(parent);                 // Add environment as dest
          out.back().index = envIndex;                      // Specify the index
        }                                      // if (envIndex != ((ulong) -1))

        std::set<ulong>::iterator t;                  // Index for active tracks
        for (t=active.begin(); t!=active.end(); ++t) // Loop over active tracks
        {
          lt.push_back(me);                     // Create a new LoseTrack message
          lt.back().set(*t, (force==RED ? BLUE : RED));  // tell of lost tracks
        }                       // for (t=active.begin(); t!=active.end(); ++t)
      }                              // node:impact[Impact:in][Destroyed:out[]]
    }                                                          // mode:Default
  }                                  // process:SensorTrack(NewtonianMotion)
}
```

C.1.86. spt/sptAngularMotion.cxx

```
////////////////////////////////////////////////////////////////////////////////
// sptAngularMotion.cxx - Class definition for the spt::AngularMotion class
//                        used to track objects.
////////////////////////////////////////////////////////////////////////////////

#include "sptAngularMotion.h"

#define END_TIME 1e307

namespace spt
```

```
{                                                          // namespace spt
  AngularMotion::AngularMotion(void)
    : p(0.0, 3), v(0.0, 3), a(0.0, 3), start(0.0), stop(END_TIME)
  {                                     // AngularMotion::AngularMotion(void)
  }                                     // AngularMotion::AngularMotion(void)

  void AngularMotion::setAM(const vertex& P,                // New acceleration
                            const vertex& V,                    // New velocity
                             const vertex& A,                    // New position
                            double t)        // Effective time of the position
  {                            // void AngularMotion::setAM(const vertex&, ...)
    update(t);                                   // Set the new effective time
    p=fixOri(P);                                       // Set the new position
    v=V;                                               // Set the new velocity
    a=A;                                           // Set the new acceleration
  }                            // void AngularMotion::setAM(const vertex&, ...)

  void AngularMotion::setAM(const vertex& P,                // New acceleration
                            const vertex& V,                    // New velocity
                             const vertex& A,                    // New position
                            double l,        // Effective time of the position
                            double u)                   // End time of motion
  {                            // void AngularMotion::setAM(const vertex&, ...)
    update(l,u);                                 // Set the new effective time
    p=fixOri(P);                                       // Set the new position
    v=V;                                               // Set the new velocity
    a=A;                                           // Set the new acceleration
  }                            // void AngularMotion::setAM(const vertex&, ...)

  void AngularMotion::ap(const vertex& P, double l)
  {                            // void AngularMotion::ap(const vertex&, double)
    update(l);                                   // Set the new effective time
    p=fixOri(P);                                       // Set the new position
  }                            // void AngularMotion::ap(const vertex&, double)

  void AngularMotion::ap(const vertex& P, double l, double u)
  {                    // void AngularMotion::ap(const vertex&, double, double)
    update(l,u);                                 // Set the new effective time
    p=fixOri(P);                                       // Set the new position
  }                    // void AngularMotion::ap(const vertex&, double, double)

  void AngularMotion::av(const vertex& V, double l)
  {                            // void AngularMotion::av(const vertex&, double)
    update(l);                                   // Set the new effective time
    v=V;                                               // Set the new velocity
  }                            // void AngularMotion::av(const vertex&, double)

  void AngularMotion::av(const vertex& V, double l, double u)
  {                    // void AngularMotion::av(const vertex&, double, double)
    update(l,u);                                 // Set the new effective time
    v=V;                                               // Set the new velocity
  }                    // void AngularMotion::av(const vertex&, double, double)

  void AngularMotion::aa(const vertex& A, double l)
  {                            // void AngularMotion::aa(const vertex&, double)
    update(l);                                   // Set the new effective time
    a=A;                                           // Set the new acceleration
  }                            // void AngularMotion::aa(const vertex&, double)

  void AngularMotion::aa(const vertex& A, double l, double u)
  {                    // void AngularMotion::aa(const vertex&, double, double)
    update(l,u);                                 // Set the new effective time
    a=A;                                           // Set the new acceleration
  }                    // void AngularMotion::aa(const vertex&, double, double)

  void AngularMotion::ap(double x, double y, double z, double l)
  {                  // void AngularMotion::ap(double, double, double, double)
    update(l);                                   // Set the new effective time
    p[0]=x; p[1]=y; p[2]=z;                            // Set the new position
    p=fixOri(p);                  // Ensure that things are in the proper ranges
  }                  // void AngularMotion::ap(double, double, double, double)
```

```
void AngularMotion::ap(double x, double y, double z, double l, double u)
{           // void AngularMotion::ap(double, double, double, double, double)
  update(l,u);                                  // Set the new effective time
  p[0]=x; p[1]=y; p[2]=z;                             // Set the new position
  p=fixOri(p);                 // Ensure that things are in the proper ranges
}           // void AngularMotion::ap(double, double, double, double, double)

void AngularMotion::av(double x, double y, double z, double l)
{                   // void AngularMotion::av(double, double, double, double)
  update(l);                                    // Set the new effective time
  v[0]=x; v[1]=y; v[2]=z;                             // Set the new velocity
}                   // void AngularMotion::av(double, double, double, double)

void AngularMotion::av(double x, double y, double z, double l, double u)
{           // void AngularMotion::av(double, double, double, double, double)
  update(l,u);                                  // Set the new effective time
  v[0]=x; v[1]=y; v[2]=z;                             // Set the new velocity
}           // void AngularMotion::av(double, double, double, double, double)

void AngularMotion::aa(double x, double y, double z, double l)
{                   // void AngularMotion::aa(double, double, double, double)
  update(l);                                    // Set the new effective time
  a[0]=x; a[1]=y; a[2]=z;                             // Set the new position
}                   // void AngularMotion::aa(double, double, double, double)

void AngularMotion::aa(double x, double y, double z, double l, double u)
{           // void AngularMotion::aa(double, double, double, double, double)
  update(l,u);                                  // Set the new effective time
  a[0]=x; a[1]=y; a[2]=z;                             // Set the new position
}           // void AngularMotion::aa(double, double, double, double, double)

vertex AngularMotion::ap(double t)
{                                         // vertex AngularMotion::ap(double)
  double dt = (min(t,stop)-start);                 // Get the time difference
  return fixOri(vertex(p+(v+0.5*a*dt)*dt));                   // Get position
}                                         // vertex AngularMotion::ap(double)

vertex AngularMotion::av(double t)
{                                         // vertex AngularMotion::av(double)
  double dt = (min(t,stop)-start);                 // Get the time difference
  return vertex(v+a*dt);                          // Get the current velocity
}                                         // vertex AngularMotion::av(double)

vertex AngularMotion::aa(double t)
{                                         // vertex AngularMotion::aa(double)
  return a;                                // Return the current acceleration
}                                         // vertex AngularMotion::aa(double)

void AngularMotion::update(double l)                  // Update to new time, l
{                                       // void AngularMotion::update(double)
  if (l>END_TIME)                 // If the start time is beyond the end time
  {
    std::cerr << "AngularMotion::update(" << l << ")" << std::endl;
    std::cerr << "Start time reduced to " << END_TIME << std::endl;
    l = END_TIME;
  }                                                        // if (l>END_TIME)
  p = ap(l);                                              // Get new position
  v = av(l);                                    // Get the new velocity vector
  start = l;                                     // Set the new effective time
  stop = END_TIME;                 // Set the end time of this leg of movement
}                                       // void AngularMotion::update(double)

void AngularMotion::update(double l, double u)        // Update to new time, l
{                               // void AngularMotion::update(double, double)
  if (l>u)                           // If start time is is beyond the end time
  {
    std::cerr << "AngularMotion::update(" << l << ", " << u << ")"
              << std::endl;
    std::cerr << "Start time reduced to " << u << std::endl;
    l = u;
    }                                                          // if (l>u)
    p = ap(l);                                            // Get new position
    v = av(l);                                 // Get the new velocity vector
    start = l;                                  // Set the new effective time
    stop = u;                       // Set the end time of this leg of movement
  }                               // void AngularMotion::update(double, double)

  double AngularMotion::getStopTime(void)
  {                                 // double AngularMotion::getStopTime(void)
    return stop;                    // Return the value to the calling routine
  }                                 // double AngularMotion::getStopTime(void)

  double AngularMotion::getStartTime(void)
  {                                // double AngularMotion::getStartTime(void)
    return start;                   // Return the value to the calling routine
  }                                // double AngularMotion::getStartTime(void)

  void AngularMotion::setStopTime(double t)                 // Update stop time
  {                                 // void AngularMotion::setStopTime(double)
    stop = t;                                             // Set new stop time
  }                                 // void AngularMotion::setStopTime(double)

  double AngularMotion::angDiff(double t1, double t2)
  {                            // double AngularMotion::angDiff(double, double)
    double rv = fixOri(t1)-fixOri(t2);                       // Get difference
    if (rv<=-PI) rv+=2.0*PI;                            // Correct to one side
    else if (rv>=PI) rv-=2.0*PI;                       // Correct to the other
    return rv;                      // Return the value to the calling routine
  }                            // double AngularMotion::angDiff(double, double)

  double AngularMotion::fixOri(double a)                        // Angle to fix
  {                                    // double AngularMotion::fixOri(double)
    a = fmod(a, 2.0*PI); if (a<0.0) a+=2.0*PI;        // Get in range [0, 2*PI)
    return a;                                         // Return the fixed value
  }                                    // double AngularMotion::fixOri(double)

  vertex AngularMotion::fixOri(vertex o)
  {                                    // vertex AngularMotion::fixOri(vertex)
    for (ulong i=0; i<o.size(); ++i) o[i]=fixOri(o[i]);          // Fix each one
    return o;                         // Return the adjusted orientation array
  }                                    // vertex AngularMotion::fixOri(vertex)
}                                                              // namespace spt
```

C.1.87. spt/sptDefs.h

```
///////////////////////////////////////////////////////////////////////////////
// sptDefs.h - Some spt namespace definitions
///////////////////////////////////////////////////////////////////////////////

#ifndef SPTDEFS_H_INCLUDED
#define SPTDEFS_H_INCLUDED

#include <valarray>

namespace spt
{                                                              // namespace spt
  typedef std::valarray<double> vertex;           // TO make life a little easier
}                                                              // namespace spt

#endif
```

C.1.88. spt/spt/Defs.cxx

```
///////////////////////////////////////////////////////////////////////////////
// sptDefs.cxx - Some simple definitions
///////////////////////////////////////////////////////////////////////////////

#include "sptDefs.h"

namespace spt
{                                                              // namespace spt
}                                                                    // namespace spt
```

C.1.89. spt/sptEnvironmentObject.h

```
////////////////////////////////////////////////////////////////////////////////
// sptEnvironmentObject.h - Class declaration for the spt::EnvironmentObject
//                          class used to sense and track objects within an
//                          environment.  It's part of namespace spt (short for
//                          'support').
////////////////////////////////////////////////////////////////////////////////

#ifndef SPTENVIRONMENTOBJECT_H_INCLUDED
#define SPTENVIRONMENTOBJECT_H_INCLUDED

#include "sptNewtonianMotion.h"
#include <string>

#define UNKNOWN 0
#define NEUTRAL 1
#define RED 2
#define BLUE 3

namespace spt
{                                                                    // namespace spt
  class EnvironmentObject : public NewtonianMotion
  {                        // class EnvironmentObject : public NewtonianMotion
    protected:
      ulong force;                             // Force to which this track belongs
      double radius;                               // Radius of sensing capability

    public:
      EnvironmentObject(void);                        // Default class constructor
      EnvironmentObject(ulong, double);                       // Class constructor
      virtual double rad(void);                           // Get the sensor radius
      virtual ulong iff(void);                            // Friend/Foe identifier
  };                       // class EnvironmentObject : public NewtonianMotion
}                                                                    // namespace spt

extern std::string forceString(ulong);

#endif
```

C.1.90. spt/sptEnvironmentObject.cxx

```
////////////////////////////////////////////////////////////////////////////////
// sptEnvironmentObject.cxx - Class definition for the spt::EnvironmentObject
//                            class used to sense objects.
////////////////////////////////////////////////////////////////////////////////

#include "sptEnvironmentObject.h"

namespace spt
{                                                                    // namespace spt
  EnvironmentObject::EnvironmentObject(void)
    : force(UNKNOWN), radius(0.0)
  {                              // EnvironmentObject::EnvironmentObject(void)
  }                              // EnvironmentObject::EnvironmentObject(void)

  EnvironmentObject::EnvironmentObject(ulong f, double r)
    : force(f), radius(r)
  {                     // EnvironmentObject::EnvironmentObject(ulong, double)
  }                     // EnvironmentObject::EnvironmentObject(ulong, double)

  double EnvironmentObject::rad(void)
  {                                          // double EnvironmentObject::rad(void)
    return radius;                                    // Return the sensor radius
  }                                          // double EnvironmentObject::rad(void)

  ulong EnvironmentObject::iff(void)
  {                                           // ulong EnvironmentObject::iff(void)
    return force;                                  // Return the force identifier
  }                                       // ulong EnvironmentObject::iff(void)
}                                                             // namespace spt

std::string forceString(ulong i)
{ return i==1 ? "NEUTRAL" : i==2 ? "RED" : i==3 ? "BLUE" : "UNKNOWN"; }
```

C.1.91. spt/sptLinearMotion.h

```
////////////////////////////////////////////////////////////////////////////////
// sptLinearMotion.h - Class declaration for the spt::LinearMotion class used
//                     to move objects in a simulation.  It's part of the spt
//                     (short for 'support') namespace.
////////////////////////////////////////////////////////////////////////////////

#ifndef SPTLINEARMOTION_H_INCLUDED
#define SPTLINEARMOTION_H_INCLUDED

#include "sptDefs.h"
#include "Trace.h"

namespace spt
{                                                             // namespace spt
  class LinearMotion : public sodl::Trace
  {                                 // class LinearMotion : public sodl::Trace
    protected:
      vertex p;                                     // Linear position vector
      vertex v;                                     // Linear velocity vector
      vertex a;                                 // Linear acceleration vector
      double start;          // Effective time of the linear motion paramters
      double stop;                                    // Boundary of movement

    public:
      LinearMotion(void);                          // Default class constructor

      virtual void setLM(const vertex&, const vertex&, const vertex&, double);
      virtual void setLM(const vertex&, const vertex&, const vertex&, double,
                         double);
      virtual void lp(const vertex&, double);                       // Set pos
      virtual void lv(const vertex&, double);                       // Set vel
      virtual void la(const vertex&, double);                       // Set acc
      virtual void lp(const vertex&, double, double);               // Set pos
      virtual void lv(const vertex&, double, double);               // Set vel
      virtual void la(const vertex&, double, double);               // Set acc
      virtual void lp(double, double, double, double);              // Set pos
      virtual void lv(double, double, double, double);              // Set vel
      virtual void la(double, double, double, double);              // Set acc
      virtual void lp(double, double, double, double, double);      // Set pos
      virtual void lv(double, double, double, double, double);      // Set vel
      virtual void la(double, double, double, double, double);      // Set acc

      virtual vertex lp(double);                           // Get the position
      virtual vertex lv(double);                           // Get the velocity
      virtual vertex la(double);                       // Get the acceleration
      virtual void update(double);                  // Update motion to time t
      virtual void update(double, double);          // Update motion to time t
      virtual double getStopTime(void);                   // Get the stop time
      virtual double getStartTime(void);                 // Get the start time
      virtual void setStopTime(double);                   // Set the stop time
  };                                // class LinearMotion : public sodl::Trace
}                                                             // namespace spt

#endif
```

C.1.92. spt/sptLinearMotion.cxx

```
///////////////////////////////////////////////////////////////////////////////
// sptLinearMotion.cxx - Class definition for the spt::LinearMotion class used
//                       to track objects.
///////////////////////////////////////////////////////////////////////////////

#include "sptLinearMotion.h"
```

```
#define END_TIME 2e307

namespace spt
{                                                              // namespace spt
  LinearMotion::LinearMotion(void)
    : p(0.0, 3), v(0.0, 3), a(0.0, 3), start(0.0), stop(END_TIME)
  {                                        // LinearMotion::LinearMotion(void)
  }                                        // LinearMotion::LinearMotion(void)

  void LinearMotion::setLM(const vertex& P,                // New acceleration
                           const vertex& V,                    // New velocity
                           const vertex& A,                    // New position
                           double l)          // Effective time of the position
  {                             // void LinearMotion::setLM(const vertex&, ...)
    update(l);                                     // Set the new effective time
    p=P;                                                 // Set the new position
    v=V;                                                 // Set the new velocity
    a=A;                                             // Set the new acceleration
  }                             // void LinearMotion::setLM(const vertex&, ...)

  void LinearMotion::setLM(const vertex& P,                // New acceleration
                           const vertex& V,                    // New velocity
                           const vertex& A,                    // New position
                           double l,          // Effective time of the position
                           double u)          // Upper bound of time in motion
  {                             // void LinearMotion::setLM(const vertex&, ...)
    update(l,u);                                   // Set the new effective time
    p=P;                                                 // Set the new position
    v=V;                                                 // Set the new velocity
    a=A;                                             // Set the new acceleration
  }                             // void LinearMotion::setLM(const vertex&, ...)

  void LinearMotion::lp(double x, double y, double z, double l)
  {                    // void LinearMotion::lp(double, double, double, double)
    update(l);                                     // Set the new effective time
    p[0]=x; p[1]=y; p[2]=z;                              // Set the new position
  }                    // void LinearMotion::lp(double, double, double, double)

  void LinearMotion::lp(double x, double y, double z, double l, double u)
  {            // void LinearMotion::lp(double, double, double, double, double)
    update(l,u);                                   // Set the new effective time
    p[0]=x; p[1]=y; p[2]=z;                              // Set the new position
  }            // void LinearMotion::lp(double, double, double, double, double)

  void LinearMotion::lv(double x, double y, double z, double l)
  {                    // void LinearMotion::lv(double, double, double, double)
    update(l);                                     // Set the new effective time
    v[0]=x; v[1]=y; v[2]=z;                              // Set the new velocity
  }                    // void LinearMotion::lv(double, double, double, double)

  void LinearMotion::lv(double x, double y, double z, double l, double u)
  {            // void LinearMotion::lv(double, double, double, double, double)
    update(l,u);                                   // Set the new effective time
    v[0]=x; v[1]=y; v[2]=z;                              // Set the new velocity
  }            // void LinearMotion::lv(double, double, double, double, double)

  void LinearMotion::la(double x, double y, double z, double l)
  {                    // void LinearMotion::la(double, double, double, double)
    update(l);                                     // Set the new effective time
    a[0]=x; a[1]=y; a[2]=z;                              // Set the new position
  }                    // void LinearMotion::la(double, double, double, double)

  void LinearMotion::la(double x, double y, double z, double l, double u)
  {            // void LinearMotion::la(double, double, double, double, double)
    update(l,u);                                   // Set the new effective time
    a[0]=x; a[1]=y; a[2]=z;                              // Set the new position
  }            // void LinearMotion::la(double, double, double, double, double)

  void LinearMotion::lp(const vertex& P, double l)
  {                              // void LinearMotion::lp(const vertex&, double)
  update(l);                                       // Set the new effective time
  p=P;                                                   // Set the new position
}                            // void LinearMotion::lp(const vertex&, double)

void LinearMotion::lp(const vertex& P, double l, double u)
{                    // void LinearMotion::lp(const vertex&, double, double)
  update(l,u);                                     // Set the new effective time
  p=P;                                                   // Set the new position
}                    // void LinearMotion::lp(const vertex&, double, double)

void LinearMotion::lv(const vertex& V, double l)
{                            // void LinearMotion::lv(const vertex&, double)
  update(l);                                       // Set the new effective time
  v=V;                                                   // Set the new velocity
}                            // void LinearMotion::lv(const vertex&, double)

void LinearMotion::lv(const vertex& V, double l, double u)
{                    // void LinearMotion::lv(const vertex&, double, double)
  update(l,u);                                     // Set the new effective time
  v=V;                                                   // Set the new velocity
}                    // void LinearMotion::lv(const vertex&, double, double)

void LinearMotion::la(const vertex& A, double l)
{                            // void LinearMotion::la(const vertex&, double)
  update(l);                                       // Set the new effective time
  a=A;                                               // Set the new acceleration
}                            // void LinearMotion::la(const vertex&, double)

void LinearMotion::la(const vertex& A, double l, double u)
{                    // void LinearMotion::la(const vertex&, double, double)
  update(l,u);                                     // Set the new effective time
  a=A;                                               // Set the new acceleration
}                    // void LinearMotion::la(const vertex&, double, double)

vertex LinearMotion::lp(double t)
{                                           // vertex LinearMotion::lp(double)
  double dt = min(t,stop)-start;                    // Get the time difference
  return vertex(p+(v+0.5*a*dt)*dt);                    // Get current position
}                                           // vertex LinearMotion::lp(double)

vertex LinearMotion::lv(double t)
{                                           // vertex LinearMotion::lv(double)
  double dt = min(t,stop)-start;                    // Get the time difference
  return vertex(v+a*dt);                          // Get the current velocity
}                                           // vertex LinearMotion::lv(double)

vertex LinearMotion::la(double t)
{                                           // vertex LinearMotion::la(double)
  return a;                                 // Return the current acceleration
}                                           // vertex LinearMotion::la(double)

void LinearMotion::update(double l)                   // Update to new time, l
{                                     // void LinearMotion::update(double)
  if (l>END_TIME)                 // If the start time is beyond the end time
  {
    std::cerr << "LinearMotion::update(" << l << ")" << std::endl;
    std::cerr << "Start time reduced to " << END_TIME << std::endl;
    l = END_TIME;
  }                                                      // if (l>END_TIME)
  p = lp(l);                                               // Get new position
  v = lv(l);                                     // Get the new velocity vector
  start = l;                                      // Set the new effective time
  stop = END_TIME;                // Set the end time of this leg of movement
}                                     // void LinearMotion::update(double)

void LinearMotion::update(double l, double u)         // Update to new time, l
{                             // void LinearMotion::update(double, double)
  if (l>u)                          // If start time is is beyond the end time
  {
    std::cerr << "LinearMotion::update(" << l << ", " << u << ")"
              << std::endl;
```

```
      std::cerr << "Start time reduced to " << u << std::endl;
      l = u;
    }                                                         // if (l>u)
    p = lp(l);                                           // Get new position
    v = lv(l);                                // Get the new velocity vector
    start = l;                                 // Set the new effective time
    stop = u;                     // Set the end time of this leg of movement
  }                             // void LinearMotion::update(double, double)

  double LinearMotion::getStopTime(void)
  {                                // double LinearMotion::getStopTime(void)
    return stop;                  // Return the value to the calling routine
  }                                // double LinearMotion::getStopTime(void)

  double LinearMotion::getStartTime(void)
  {                               // double LinearMotion::getStartTime(void)
    return start;                 // Return the value to the calling routine
  }                               // double LinearMotion::getStartTime(void)

  void LinearMotion::setStopTime(double t)               // Update stop time
  {                                // void LinearMotion::setStopTime(double)
    stop = t;                                           // Set new stop time
  }                                // void LinearMotion::setStopTime(double)
}                                                           // namespace spt
```

C.1.93. spt/sptNewtonianMotion.h

```
////////////////////////////////////////////////////////////////////////////////
// sptNewtonianMotion.h - Class declaration for the spt::NewtonianMotion class
//                        used to move objects in a simulation.  It's part of
//                        the spt (short for 'support') namespace.
////////////////////////////////////////////////////////////////////////////////

#ifndef SPTNEWTONIANMOTION_H_INCLUDED
#define SPTNEWTONIANMOTION_H_INCLUDED

#include "sptAngularMotion.h"
#include "sptLinearMotion.h"

namespace spt
{                                                           // namespace spt
  class NewtonianMotion : public AngularMotion, public LinearMotion
  {        // class NewtonianMotion : public AngularMotion, public LinearMotion
    public:
      NewtonianMotion(void);                     // Default class constructor
      virtual void set(const vertex&, const vertex&, const vertex&,
                       const vertex&, const vertex&, const vertex&, double);

      virtual void set(const vertex&, const vertex&, const vertex&,
                       const vertex&, const vertex&, const vertex&, double,
                       double);

      virtual void update(double);           // Update motion params to time t
      virtual void update(double, double);   // Update motion params to time t
      virtual double getStopTime(void);                     // Get the stop time
      virtual double getStartTime(void);                   // Get the start time
      virtual void setStopTime(double);                     // Set the stop time
  };       // class NewtonianMotion : public AngularMotion, public LinearMotion
}                                                           // namespace spt

#endif
```

C.1.94. spt/sptNewtonianMotion.cxx

```
////////////////////////////////////////////////////////////////////////////////
// sptNewtonianMotion.cxx - Class definition for the spt::NewtonianMotion
//                          class used to track objects.
////////////////////////////////////////////////////////////////////////////////

#include "sptNewtonianMotion.h"
```

```
namespace spt
{                                                                  // namespace spt
  NewtonianMotion::NewtonianMotion(void)
  {                                    // NewtonianMotion::NewtonianMotion(void)
  }                                    // NewtonianMotion::NewtonianMotion(void)

  void NewtonianMotion::set(const vertex& LP,     // New lin acc
                            const vertex& LV,     // New lin vel
                            const vertex& LA,     // New lin pos
                            const vertex& AP ,    // New ang acc
                            const vertex& AV,     // New ang vel
                            const vertex& AA,     // New ang pos
                            double l)         // Effective time of the position
  {             // void NewtonianMotion::set(const vertex&, ...)
    update(l);                        // Update to the current motion parameters
    LinearMotion::p=LP;                                    // Set the new position
    LinearMotion::v=LV;                                    // Set the new velocity
    LinearMotion::a=LA;                                // Set the new acceleration
    AngularMotion::p=fixOri(AP);                   // Set the new angular position
    AngularMotion::a=AA;                       // Set the new angular acceleration
    AngularMotion::v=AV;                           // Set the new angular velocity
  }             // void NewtonianMotion::set(const vertex&, ...)

  void NewtonianMotion::set(const vertex& LP,     // New lin acc
                            const vertex& LV,     // New lin vel
                         const vertex& LA,     // New lin pos
                         const vertex& AP,     // New ang acc
                         const vertex& AV,     // New ang vel
                         const vertex& AA,     // New ang pos
                         double l, double u)       // Effective motion times
  {             // void NewtonianMotion::set(const vertex&, ...)
    update(l,u);                      // Update to the current motion parameters
    LinearMotion::p=LP;                                    // Set the new position
    LinearMotion::v=LV;                                    // Set the new velocity
    LinearMotion::a=LA;                                // Set the new acceleration
    AngularMotion::p=fixOri(AP);                   // Set the new angular position
    AngularMotion::a=AA;                       // Set the new angular acceleration
    AngularMotion::v=AV;                           // Set the new angular velocity
  }             // void NewtonianMotion::set(const vertex&, ...)

  void NewtonianMotion::update(double l)             // Update to new time, t
  {                                      // void NewtonianMotion::update(double)
    LinearMotion::update(l);        // Set new effective time for linear motion
    AngularMotion::update(l);      // Set new effective time for angular motion
  }                                      // void NewtonianMotion::update(double)

  void NewtonianMotion::update(double l, double u)     // Update to new time, l
  {                                      // void NewtonianMotion::update(double)
    LinearMotion::update(l,u);      // Set new effective time for linear motion
    AngularMotion::update(l,u);    // Set new effective time for angular motion
  }                                      // void NewtonianMotion::update(double)

  double NewtonianMotion::getStopTime(void)
  {                                 // double NewtonianMotion::getStopTime(void)
    return min(LinearMotion::getStopTime(), AngularMotion::getStopTime());
  }                                 // double NewtonianMotion::getStopTime(void)

  double NewtonianMotion::getStartTime(void)
  {                                // double NewtonianMotion::getStartTime(void)
    return max(LinearMotion::getStartTime(), AngularMotion::getStartTime());
  }                                // double NewtonianMotion::getStartTime(void)

  void NewtonianMotion::setStopTime(double t)                // Update stop time
  {                                 // void NewtonianMotion::setStopTime(double)
    LinearMotion::setStopTime(t);          // Set new stop time for linear motion
    AngularMotion::setStopTime(t);        // Set new stop time for angular motion
  }                                 // void NewtonianMotion::setStopTime(double)
}                                                                  // namespace spt
```

C.2. Bounce1

C.2.1. bounce.proc

```
{import process {View3D, Node3D, Cube, Polygon3D, particle} }
{import message {start, gr_update, AddNode3D, AddShape3D,
                 SetColor, SetMode, SetPosition, SetSize,
                 SetCubeSize, SetRefresh, set_system, SetPointSize,
                 StartSimulation} }
{import {<stdlib.h>, <time.h>, <GL/glut.h>} }
{import std {<iostream>}}

{
  process:bounce
  {
    double:interval(0.025);                          // Interval between updates
    particle:b[200];                         // Collection of bouncing particles
    View3D:view;                                          // Display to this view
    Node3D:systemNode;                                    // Node for the display
    Polygon3D:system;                          // Place where the points are stored
    Cube:cube;                   // Cube for surrounding the collection of points

    mode:Default
    {                                                              // mode:Default
      node:start_sim[StartSimulation:strt]          // Initial Start message
                     [start:s=>(b;):(0.0),       // Transmit start to particles
                      set_system:setSystem=>(b;),          // Set parent system
                      AddNode3D:an=>(view;),         // Add system node to view
                      AddShape3D:as=>(systemNode;),        // Add shape to node
                      SetColor:scSystem=>(system;),        // Set system color
                      SetColor:scCube=>(cube;),              // Set cube color
                      SetMode:smSystem=>(system;),          // Set system mode
                      SetMode:smCube=>(cube;),                // Set cube mode
                      SetPosition:sp=>(view;):(0.0),       // Set view position
                      SetSize:ss=>(view;):(0.0),               // Set view size
                      SetRefresh:sr=>(view;),              // Set refresh rate
                      SetCubeSize:scs=>(cube;),                // Set cube size
                      gr_update:up=>(me;b;):(interval-1e-9),        // Update #1
                      SetPointSize:sps=>(view;)]              // Set point size
      {                // node:start_sim[StartSimulation:strt][start:s, ... ]
        sr.set(interval);                          // Set the refresh interval
        setSystem.system = system;         // Set the system for the particles
        an.add(systemNode);                          // Main node to add to the view
        as.add(system);                    // System of particles to add to node
        as.add(cube);                     // Cube exterior to the particle system
        scSystem.set(0.0, 1.0, 1.0);                 // Set system color to cyan
        scCube.set(0.0, 1.0, 0.0);                    // Set cube color to green
        smSystem.gr_mode = smCube.gr_mode = GL_POINTS;         // Rendering mode
        sp.set(50, 50);                        // Set the position of the view
        ss.set(800,600);                         // Set the size of the window
        scs.size = 20.0;                           // Set the size of the cube
        sps.size = 3.0;                             // Specify the point size
      }                // node:start_sim[StartSimulation:strt][start:s, ... ]

      node:update[gr_update:in]
                  [gr_update:out=>(me; b;):(getTime()+interval)]
      { }
    }                                                              // mode:Default
  }                                                             // process:bounce
}
```

C.2.2. gr_update.msg

```
{message:gr_update;}
```

C.2.3. hit.msg

```
{
  message:hit
  {                                                        // message:hit
    int:axis;                         // Axis associated with the hit event
  }                                                        // message:hit
}
```

C.2.4. particle.proc

```
{import message {hit, start, gr_update, set_system, SetVertex3D,
                 AddVertex3D} }
{import process {Node3D, Vertex3D} }
{import std {<vector>} }
{import {"Exception.h"} }

{
  process:particle
  {
    Vertex3D:vrt;                                        // Screen vertex
    double:pos[3];                                     // Position vector
    double:vel[3];                                     // Velocity vector
    double:nextTime[3];                     // Next impact times for each axis
    double:time(0.0);                     // Time for the last velocity change
    sodl::Defs::MessageType:lm(SMT_LAST);                // Last message type

    method:init(public; void;)
    {                                           // method:init(public; void;)
      for (uint i=0; i<3; ++i)            // Loop over each of the coordinates
      {
        pos[i] = random.nextDouble(-10.0, 10.0);           // Assign position
        vel[i] = random.nextDouble(-10.0, 10.0);           // Assign position
        setNextHitTime(i);            // Set time for next impact along axis i
      }                                            // for (int i=0; i<3; ++i)
    }                                           // method:init(public; void;)

    method:setNextHitTime(private; void; int:i;)
    {                      // method:setNextHitTime(private; void; int:i;)
      if (vel[i]<0.0) nextTime[i] = time-(10.0+pos[i])/vel[i];
      else if (vel[i]>0.0) nextTime[i] = time+(10.0-pos[i])/vel[i];
      else nextTime[i]=getEngine().getClock().getEndTime();        // vel[i]==0
    }                      // method:setNextHitTime(private; void; int:i;)

    method:move(private; void;)
    {                             // method:move(private; void; double time)
      double dt = getTime()-time;                               // Delta time
      for (uint i=0; i<3; ++i) pos[i] += dt*vel[i];          // Update position
      time = getTime();                                      // Update the time
    }                             // method:move(private; void; double time)

    method:getMinAxis(private; int;)
    {                                     // method:getMinAxis(private; int;)
      int axis = 0;                              // Initialize the axis value

      for (uint i=1; i<nextTime.size(); ++i)              // Loop over the axes
        if (nextTime[axis]>nextTime[i]) axis=i;                 // Get min time
      return axis;                                   // Return that axis value
    }                                     // method:getMinAxis(private; int;)

    mode:Default
    {                                                        // mode:Default
      node:setSystem[set_system:in][AddVertex3D:av=>(in.system;)]
      {                 // node:setSystem[set_system:in][AddVertex3D:out]
        lm = in.getType();
        av.add(vrt);                             // Add the vertex to the system
      }                 // node:setSystem[set_system:in][AddVertex3D:out]

      node:start_sim[start:s]
                    [hit:out=>(me;):(nextTime[out.axis])]
      {                           // node:start_sim[start:s][hit:out=>(me;)]
        lm = s.getType();
        out.axis = getMinAxis();                       // Axis for the impact
      }                           // node:start_sim[start:s][hit:out=>(me;)]
      node:update[gr_update:in][SetVertex3D:out=>(vrt;)]
      {                           // node:update[gr_update:in][SetVertex3D:out]
        lm = in.getType();
        move();                    // Move the particle to the current position
        out.set(pos);                             // Update the vertex position
      }                           // node:update[gr_update:in][SetVertex3D:out]

      node:change[hit:in]
                 [hit:out=>(me;):(nextTime[out.axis])]
      {                                  // node:change[hit:in][hit:out=>(me;)]
        lm = in.getType();
        move();                    // Move the particle to the current position
        vel[in.axis] = -vel[in.axis];                    // Change the velocity
        setNextHitTime(in.axis);        // Set next hit time for specified axis
        out.axis = getMinAxis();                         // Axis for the impact
      }                                  // node:change[hit:in][hit:out=>(me;)]
    }                                                          // mode:Default
  }                                                        // process:particle
}
```

C.2.5. set_system.msg

```
{
  message:set_system
  {                                                    // message:set_system
    process:system;                 // System to which particles will be added
  }                                                    // message:set_system
}
```

C.2.6. start.msg

```
{ message:start; }
```

C.3. Bounce2

C.3.1. bounce.proc

```
{import process {bounce_view, Node3D, Cube, Polygon3D, particle} }
{import message {AddNode3D, AddShape3D, SetColor, SetMode, SetSize,
                 SetPosition, SetCubeSize, SetRefresh, SetPointSize,
                 StartSimulation, AddVertex3D} }
{import {<stdlib.h>, <time.h>, <GL/glut.h>} }
{import std {<iostream>}}

{
  process:bounce
  {
    particle:b[2000];                       // Collection of bouncing particles
    bounce_view:view;                                // Display to this view
    Node3D:systemNode;                               // Node for the display
    Polygon3D:system;                      // Place where the points are stored
    Cube:cube;                  // Cube for surrounding the collection of points

    mode:Default
    {                                                          // mode:Default
      node:start_sim[StartSimulation:strt]               // Initial start message
                    [AddNode3D:an=>(view;),             // Add system node to view
                     AddShape3D:as=>(systemNode;),            // Add shape to node
                     AddVertex3D:av=>(system;),              // Add vert to system
                     SetColor:scSystem=>(system;),             // Set system color
                     SetColor:scCube=>(cube;),                   // Set cube color
                     SetMode:smSystem=>(system;),            // Set the system mode
                     SetMode:smCube=>(cube;),                  // Set the cube mode
                     SetPosition:sp=>(view;),              // Set the view position
                     SetSize:ss=>(view;),                      // Set the view size
                     SetRefresh:sr=>(view;),                // Set the refresh rate
                     SetCubeSize:scs=>(cube;),                 // Set the cube size
                     SetPointSize:sps=>(view;)]               // Set the point size
      {              // node:start_sim[StartSimulation:strt][start:s, ... ]
```

```
        an.add(systemNode);                        // Main node to add to the view
        as.add(system);                      // System of particles to add to node
        as.add(cube);                      // Cube exterior to the particle system
        for (uint i=0; i<b.size(); ++i) av.add(b[i]);          // Add particles
        scSystem.set(0.0, 1.0, 1.0);                // Set system color to cyan
        scCube.set(0.0, 1.0, 0.0);                   // Set cube color to green
        smSystem.gr_mode=smCube.gr_mode=GL_POINTS;          // Rendering mode
        sp.set(50, 50);                            // Set the position of the view
        ss.set(800,600);                            // Set the size of the window
        sr.set(0.025);                               // Set the refresh interval
        scs.size = 20.0;                             // Set the size of the cube
        sps.size = 3.0;                               // Specify the point size
      }                 // node:start_sim[StartSimulation:strt][start:s, ... ]
    }                                                         // mode:Default
  }                                                         // process:bounce
}
```

C.3.2. bounce_view.proc

```
{import process {View3D} }
{import message {set_motion} }
{import gvm {gvmBounceView, gvmParticle, gvmSetMotion} }

{
  process:bounce_view(View3D)
  {
    method:init(public; void;)
    {                                              // method:init(public; void;)
      view = new gvm::BounceView;                         // Create a new view
      dynamic_cast<GLUTViewManager&>(*EngineStand::stand.vm).
                    addView(view);
    }                                              // method:init(public; void;)

    method:getGVMType(protected; gvm::object_type; ptype:t;)
    {                  // method:getGVMType(protected; gvm::object_type; ptype;)
      switch(t)                                            // Which one is it?
      {
        case SPT_particle: return gvm::GVM_Particle;
        default: return View3D::getGVMType(t);
      }                                                           // switch(t)
    }                  // method:getGVMType(protected; gvm::object_type; ptype;)

    mode:Default
    {                                                          // mode:Default
      node:setMotion[set_motion:in][]
      {                                       // node:setMotion[set_motion:in][]
       view->schedule(new gvm::SetMotion(*view, getTime(), in.index,
                                  in.getT(),in.getP(), in.getV(),
                                  in.getA()));
      }                                       // node:setMotion[set_motion:in][]
    }                                                          // mode:Default
  }
}
```

C.3.3. hit.msg

```
{
  message:hit
  {                                                         // message:hit
    int:axis;                         // Axis associated with the hit event
  }                                                         // message:hit
}
```

C.3.4. particle.proc

```
{import message {hit, set_motion, SetVertex3D, StartSimulation,
                 AddVertex3D, AddView} }
{import process {Vertex3D} }
{import std {<vector>} }
{import {"Exception.h"} }
```

```
{
  process:particle(Vertex3D)
  {
    double:vel[3];                                              // Velocity vector
    double:acc[3];                                          // Acceleration vector
    double:nextTime[3];                             // Next impact times for each axis
    double:time(0.0);                             // Time for the last velocity change

    method:init(public; void;)
    {                                                   // method:init(public; void;)
      for (uint i=0; i<3; ++i)                    // Loop over each of the coordinates
      {
        pos[i] = random.nextDouble(-10.0, 10.0);                  // Assign position
        vel[i] = random.nextDouble(-10.0, 10.0);                  // Assign velocity
        acc[i] = (i==2) ? -9.8 : 0.0;                     // Initialize acceleration
        setNextHitTime(i);                  // Set time for next impact along axis i
      }                                                   // for (int i=0; i<3; ++i)
    }                                                   // method:init(public; void;)

    method:setNextHitTime(private; void; int:i;)
    {                                  // method:setNextHitTime(private; void; int:i;)
      static double et = getEngine().getClock().getEndTime();
      static double s = 10.0;

      if (acc[i] != 0.0)                          // If there is non-zero acceleration
      {
        double a = acc[i]/2.0;
        double a2 = acc[i];
        double b = vel[i];
        double bs = b*b;
        double c = pos[i];
        double pd = bs-4.0*(c-s)*a;                                 // Discriminant 1
        double nd = bs-4.0*(c+s)*a;                                 // Discriminant 2
        double psr = (pd>=0.0) ? sqrt(pd) : -1.0;
        double nsr = (nd>=0.0) ? sqrt(nd) : -1.0;
        double pt = (pd>0.0) ? min((-b+psr)/a2, (-b-psr)/a2) : et;
        double nt = (nd>0.0) ? max((-b+nsr)/a2, (-b-nsr)/a2) : et;
        if (pt<0.0 && nt<0.0)
          throw Exception::Nonspecific("Bounce fault");
        else if (pt<0.0 || nt<0.0) nextTime[i] = time+max(pt, nt);
        else nextTime[i]=time+min(pt, nt);                 // It should hit one wall
      }                                                       // if (acc[i] != 0.0)
      else if (vel[i]<0.0) nextTime[i] = time-(10.0+pos[i])/vel[i];
      else if (vel[i]>0.0) nextTime[i] = time+(10.0-pos[i])/vel[i];
      else nextTime[i]=getEngine().getClock().getEndTime();              // vel[i]==0
    }                                  // method:setNextHitTime(private; void; int:i;)

    method:move(private; void;)
    {                                       // method:move(private; void; double time)
      double dt = getTime()-time;                                       // Delta time
      double ddt = dt*dt;                                         // Get the delta time
      for (uint i=0; i<3; ++i)                                    // Loop over the axes
      {
        pos[i] += dt*vel[i]+acc[i]*ddt*0.5;               // Get the current position
        vel[i] += dt*acc[i];                              // Get the current velocity
      }                                                  // for (uint i=0; i<3; ++i)
      time = getTime();                                            // Update the time
    }                                       // method:move(private; void; double time)

    method:getMinAxis(private; int;)
    {                                               // method:getMinAxis(private; int;)
      int axis = 0;                                       // Initialize the axis value

      for (uint i=1; i<nextTime.size(); ++i)                      // Loop over the axes
        if (nextTime[axis]>nextTime[i]) axis=i;                         // Get min time
      return axis;                                           // Return that axis value
    }                                               // method:getMinAxis(private; int;)

    mode:Default
    {                                                                  // mode:Default
```

```
      node:start[StartSimulation:s]
                [hit:out=>(me;):(nextTime[out.axis])]
      {                       // node:start[StartSimulation:s][hit:out=>(me;)]
        out.axis = getMinAxis();                         // Axis for the impact
      }                       // node:start[StartSimulation:s][hit:out=>(me;)]

      node:addView[AddView:in][set_motion:out=>(in.getSource();)]
      {          // node:addView[AddView:in][set_motion:out=>(in.getSource();)]
        out.set(time,pos,vel,acc);                // Set parameters in new view
        out.index = in.index;                      // Set the index value, too
      }          // node:addView[AddView:in][set_motion:out=>(in.getSource();)]

      node:change[hit:in]
                 [hit:out=>(me;):(nextTime[out.axis]),
                  set_motion:sm[]]
      {                                 // node:change[hit:in][hit:out=>(me;)]
        move();                     // Move the particle to the current position
        vel[in.axis] = -vel[in.axis];                     // Change the velocity
        setNextHitTime(in.axis);        // Set next hit time for specified axis
        out.axis = getMinAxis();                         // Axis for the impact

        std::map<process, gvm::object_index>::iterator i;          // For index
        for(i=views.begin(); i!=views.end(); ++i)         // Loop over index map
        {
          sm.push_back(me);                                     // Make new msg
          sm.back().addDest(i->first);                // Add a destination to it
          sm.back().index = i->second;                      // Specify the index
          sm.back().set(getTime(),pos,vel,acc);             // Specify motion
        }                           // for (i=views.begin(); i!=views.end(); ++i)
      }                                 // node:change[hit:in][hit:out=>(me;)]
    }                                                          // mode:Default
  }                                                         // process:particle
}
```

C.3.5. set_motion.msg

```
{import message {SetValue} }
{import std {<vector>} }
{
  message:set_motion(SetValue)
  {                                             // message:set_motion(SetValue)
    double:t;                                // Effective time of these settings
    double:a[3];                                       // Acceleration to set
    double:p[3];                                           // Position to set
    double:v[3];                                           // Velocity to set

    method:set(public; void; double:T;                        // Effective time
                         std::vector<double>:P;                 // Position at T
                         std::vector<double>:V;                 // Velocity at T
                         std::vector<double>:A;)// Acceleration at T
    {                              // method:set(public; void; doubleT; ... )
      t=T;                                             // Set the effective time
      p=resize_vector(3, 0.0, P);              // Set the position at that time
      v=resize_vector(3, 0.0, V);              // Set the velocity at that time
      a=resize_vector(3, 0.0, A);          // Set the acceleration at that time
    }                              // method:set(public; void; doubleT; ... )

    method:getT(public; double;) { return t; }        // Get the effective time
    method:getP(public; std::vector<double>;) { return p; }          // Get pos
    method:getV(public; std::vector<double>;) { return v; }          // Get vel
    method:getA(public; std::vector<double>;) { return a; }          // Get acc
  }                                             // message:set_motion(SetValue)
}
```

C.3.6. gvm/gvmBounceView.h

```
#ifndef GVMBOUNCEVIEW_H_INCLUDED
#define GVMBOUNCEVIEW_H_INCLUDED

#include "gvmView3D.h"
```

```
namespace gvm
{                                                               // namespace gvm
  class BounceView : public View3D
  {                                        // class BounceView : public View3D
    public:
      BounceView(void);                                   // Class constructor

      virtual bool isDisplay(void);          // Should we update the display?
      virtual void createObject(object_handle);      // Create object to view
  };                                       // class BounceView : public View3D
}                                                               // namespace gvm
#endif
```

C.3.7. gvm/gvmBounceView.cxx

```
#include <math.h>

#include "Exception.h"
#include "gvmBounceView.h"
#include "gvmParticle.h"

namespace gvm
{                                                               // namespace gvm
  BounceView::BounceView(void)
  {                                           // BounceView::BounceView(void)
    setTitle("BounceView");
  }                                           // BounceView::BounceView(void)

  bool BounceView::isDisplay(void)
  {                                        // bool BounceView::isDisplay(void)
    return (isVisible && refresh);                     // Should we redisplay
  }                                        // bool BounceView::isDisplay(void)

  void BounceView::createObject(object_handle h)
  {                          // void BounceView::createObject(object_handle)
    if (h.second >= objectList.size())            // Is this lined up properly
      throw Exception::Nonspecific("Object count mis-alignment.");

    switch (h.first)                          // Which object should we create?
    {
      case GVM_Particle:                   // A new Particle instance requested
        objectList[h.second] = new Particle(*this, h.second);
        break;                                            // case GVM_Particle:
      default:                                              // None of the above
        View3D::createObject(h);                     // Create a default object
        break;                                                         // default
    }                                                              // switch (t)
  }                // void BounceView::createObject(object_type, object_index)
}                                                               // namespace gvm
```

C.3.8. gvm/gvmParticle.h

```
#ifndef GVMPARTICLE_H_INCLUDED
#define GVMPARTICLE_H_INCLUDED

#include <vector>
#include "gvmVertex3D.h"

#define GVM_Particle GVM_UserObject000

namespace gvm
{
  class BounceView;

  class Particle : public Vertex3D
  {
    protected:
      double t;
      std::vector<double> pos;
      std::vector<double> vel;
      std::vector<double> acc;
    protected:
      Particle(gvm::BounceView&,object_type,ulong);        // Class constructor

    public:
      Particle(gvm::BounceView&,ulong);                    // Class constructor

      virtual void setMotion(double, const std::vector<double>&,
                             const std::vector<double>&,
                             const std::vector<double>&);
      virtual void display(void);                      // Display the component
      virtual bool isType(object_type);           // Check if this is of type t
  };                                                         // class Particle
}                                                             // namespace gvm
#endif
```

C.3.9. gvm/gvmParticle.cxx

```
#include <iostream>
#include "gvmParticle.h"
#include "gvmBounceView.h"

namespace gvm
{
  Particle::Particle(gvm::BounceView& v, object_type t, ulong i)
    : Vertex3D(v, t, i), t(0.0), pos(3,0.0), vel(3,0.0), acc(3,0.0)
  {                  // Particle::Particle(gvm::BounceView&, object_type, ulong)
  }                  // Particle::Particle(gvm::BounceView&, object_type, ulong)

  Particle::Particle(gvm::BounceView& v, ulong i)
    : Vertex3D(v,GVM_Particle,i), t(0.0), pos(3,0.0), vel(3,0.0),
      acc(3,0.0)
  {                               // Particle::Particle(gvm::BounceView&, ulong)
  }                               // Particle::Particle(gvm::BounceView&, ulong)

  void Particle::setMotion(double T,
                           const std::vector<double>& P,
                            const std::vector<double>& V,
                            const std::vector<double>& A)
  {                                         // void Particle::set(double, ... )
    t=T;                            // Set the effective time for the parameters
    pos=P;             // Set the position of the particle at the effective time
    vel=V;             // Set the velocity of the particle at the effective time
    acc=A;         // Set the acceleration of the particle at the effective time
  }                                         // void Particle::set(double, ... )

  void Particle::display(void)
  {                                             // void Particle::display(void)
    double dt = getView().getTime()-t;           // Getdelta since last update
    for (uint i=0; i<3; ++i)                      // Loop over each coordinate
      loc[i]=pos[i]+(vel[i]+0.5*acc[i]*dt)*dt;       // Get new point location
    Vertex3D::display();                                  // Display the vertex
  }                                             // void Particle::display(void)

  bool Particle::isType(object_type c)
  {                                       // bool Particle::isType(object_type)
    return (c==GVM_Particle || Vertex3D::isType(c));          // Return results
  }                                       // bool Particle::isType(object_type)
}                                                             // namespace gvm
```

C.3.10. gvm/gvmSetMotion.h

```
#ifndef SETMOTION_H_INCLUDED
#define SETMOTION_H_INCLUDED

#include "gvmMessage.h"
#include "gvmObject.h"

#define GVM_SetMotion GVM_UserMessage000

namespace gvm
```

```
{                                                      // namespace gvm
  class View;                   // Forward declaration of the gvm::View class

  class SetMotion : public Message
  {                                     // class SetMotion : public Message
    private:
      double t;                 // Effective time stamp of the motion parameters
      std::vector<GLdouble> p;                         // Position at time t
      std::vector<GLdouble> v;                         // Velocity at time t
      std::vector<GLdouble> a;                     // Acceleration at time t

    public:
      SetMotion(View&, double, object_index, double,
                std::vector<GLdouble>, std::vector<GLdouble>,
                std::vector<GLdouble>);

      virtual void send(void);                  // Deliver the message payload
  };                                    // class SetMotion : public Message
}                                                      // namespace gvm
#endif
```

C.3.11. gvm/gvmSetMotion.cxx

```
#include "Exception.h"
#include "gvmParticle.h"
#include "gvmSetMotion.h"
#include "gvmView.h"

namespace gvm
{                                                      // namespace gvm
  SetMotion::SetMotion(View& v,         // Parent (owning) view of this message
                       double t,                 // Time to process the message
                       object_index i,                  // Destination object
                       double T,         // Effective time of motion parameters
                       std::vector<GLdouble> P,         // Position at time T
                       std::vector<GLdouble> V,         // Velocity at time T
                       std::vector<GLdouble> A)          // Acceleration at T
    : Message(v, t, GVM_SetMotion, i), t(T), p(P), v(V), a(A)
  {                    // SetMotion::SetMotion(View&, double, object_index, ... )
  }                    // SetMotion::SetMotion(View&, double, object_index, ... )

  void SetMotion::send(void)
  {                                            // void SetMotion::send(void)
    gvm::Particle *part =
          dynamic_cast<gvm::Particle*>(&getView()[getDest()]);
    if (part==NULL)
      throw Exception::BadCast("gvm::Object", "gvm::Particle");
    part->setMotion(t,p,v,a);           // Set motion parameters of destination
  }                                            // void SetMotion::send(void)
}                                                      // namespace gvm
```

C.4. Brigade1

C.4.1. battalion.proc

```
{import process {unit, company} }
{import message {StartSimulation, SetColor} }

{
  process:battalion(unit)
  {                                             // process:battalion(unit)
    company:subordinates[4];                        // Subordinate objects

    method:init(protected; void;)
    {                                     // method:init(protected; void;)
      unit::init();                             // Call parent class version
      subs = subordinates;                 // Copy the subordinate handles
      sub_labels.push_back(" Alpha Company");
      sub_labels.push_back(" Bravo Company");
      sub_labels.push_back(" Charlie Company");
      sub_labels.push_back(" Delta Company");
    }                                           // method:init(protected; void;)
  }
}
```

C.4.2. brigade.proc

```
{import process {unit, battalion, View2D, Polygon2D, Vertex2D} }
{import message {order, set_parent, StartSimulation, SetColor,
                 AddVertex2D, SetVertex2D, AddNode2D, SetMode, SetSize,
                 SetPosition, SetRefresh} }
{import std {<string>, <iostream>} }

{
  process:brigade(unit)
  {
    battalion:subordinates[4];                       // Subordinate battalions
    View2D:view;                                    // View where stuff is seen
    Polygon2D:rect;                          // A rectangle everbody will reference
    Vertex2D:vert[4];                               // The vertices of the rectangle

    method:init(protected; void;)
    {                                           // method:init(protected; void;)
      unit::init();                                  // Call parent class version
      subs = subordinates;                      // Copy the subordinate handles
      sub_labels.push_back(" 1st Battalion");
      sub_labels.push_back(" 2nd Battalion");
      sub_labels.push_back(" 3rd Battalion");
      sub_labels.push_back(" 4th Battalion");
    }                                           // method:init(protected; void;)

    method:getScale(public; std::vector<double>;)
    { return make_vector(2, 1.0, 1.0); }

    method:getTranslation(public; std::vector<double>;)
    { return make_vector(2, 0.0, 0.0); }

    mode:start
    {                                                            // mode:start
      node:start[StartSimulation:strt]                      // Initial message
                [order:out=>(me;):(0.0),                 // Start sim at time 0
                 set_parent:sp=>(me;),          // Set subordinate parent handle
                 AddVertex2D:av=>(rect;),            // Add vertex loc to shape
                 SetVertex2D:sv[4]=>(vert[@];),        // Set vertex locations
                 AddNode2D:an=>(view;),              // Add the node to the view
                 SetMode:sm=>(rect;),                 // Set the rectangle mode
                 SetSize:ss=>(view;),                        // Set window size
                 SetPosition:spos=>(view;),              // Set window position
                 SetRefresh:srfsh=>(view;)]             // Set refresh interval
      {                               // node:start[StartSimulation:strt][...]
        sp.instance = 0;                               // Set the instance number
        sp.label = "Brigade";                          // Set this instance label
        sp.parent_node = unit_node;                         // Set the unit handle
        sp.rect = rect;                               // Set the rectangle handle

        sm.set(GL_QUADS);                                    // Filled rectangle

        out.data = 5.0+random.nextDouble(5.0);           // Generate data value

        an.add(unit_node);          // Add brigade's node as root node in view

        sv[0].set(-50.0, -0.5);                                   // First vertex
        sv[1].set( 50.0, -0.5);                                  // Second vertex
        sv[2].set( 50.0,  0.5);                                   // Third vertex
        sv[3].set(-50.0,  0.5);                                  // Fourth vertex
        for (int i=0; i<4; ++i)                          // Loop over the vertices
          av.add(vert[i]);                                // Specify the vertices

        ss.set(800,600);                    // Set the viewport size to 800x600
        spos.set(100,100);                      // And positioned at (100,100)
        srfsh.set(0.1);                   // Set it to refresh every 0.5 time unit
      }                                   // node:start[StartSimulation:strt][...]
    }                                                              // mode:start
  }
}
```

C.4.3. company.proc

```
{import process {unit, platoon} }
{import std {<string>, <iostream>} }
{import message {StartSimulation, SetColor} }

{
  process:company(unit)
  {                                                 // process:company(unit)
    platoon:subordinates[4];

    method:init(protected; void;)
    {                                       // method:init(protected; void;)
      unit::init();                             // Call parent class version
      subs = subordinates;                   // Copy the subordinate handles
      sub_labels.push_back(" 1st Platoon");       // Subordinate label values
      sub_labels.push_back(" 2nd Platoon");
      sub_labels.push_back(" 3rd Platoon");
      sub_labels.push_back(" 4th Platoon");
    }                                       // method:init(protected; void;)
  }                                                 // process:company(unit)
}
```

C.4.4. order.msg

```
{
  message:order
  {
    double:data(0.0);
  }
}
```

C.4.5. platoon.proc

```
{import process {unit, squad} }
{import std {<string>, <vector>, <iostream>} }
{import message {StartSimulation, SetColor} }

{
  process:platoon(unit)
  {                                                 // process:platoon(unit)
    squad:subordinates[4];

    method:init(protected; void;)
    {                                       // method:init(protected; void;)
      unit::init();                             // Call parent class version
      subs = subordinates;                   // Copy the subordinate handles
      sub_labels.push_back(" Squad 1");           // Subordinate label values
      sub_labels.push_back(" Squad 2");
      sub_labels.push_back(" Squad 3");
      sub_labels.push_back(" Squad 4");
    }                                       // method:init(protected; void;)
  }                                                 // process:platoon(unit)
}
```

C.4.6. report.msg

```
{
  message:report
  {
    long:instance(0);
  }
}
```

C.4.7. set_parent.msg

```
{import std {<string>} }

{
  message:set_parent
  {                                                          // message:set_parent
    int:instance;           // Subordinate instance number of the destination
    std::string:label;                            // Label of the destination
    process:parent_node;                                // Node for the parent
    process:rect;                             // Polygon for the figure to use
  }                                                          // message:set_parent
}
```

C.4.8. soldier.proc

```
{import process {unit} }
{import message {order, report, SetColor, StartSimulation} }
{import std {<string>, <iostream>} }

{
  process:soldier(unit)
  {
    method:init(protected; void;)
    {                                         // method:init(protected; void;)
      unit::init();                                // Call parent class version
      subs.push_back(me);                       // Copy the subordinate handles
    }                                         // method:init(protected; void;)

    method:fossilCollect(protected; void;)
    {                   // method:statusReport(protected; void; sim:report&:in;)
      if (getTime() >= 0.0)                 // If the timestamp is not negative
      {
        if (waiting_for_orders.isActive())             // If waiting for orders
        {
          std::cout << getTime() << ":" << std::endl;
          std::cout << getLabel() << " following orders: "
                    << sub_times[0] << std::endl << std::endl;
        }                                  // if (waiting_for_orders.isActive())
        else                                           // If it's anything else
          unit::fossilCollect();        // Let the parent class take care of it
      }                                                // if (getTime() >= 0.0)
    }                   // method:statusReport(protected; void; sim:report&:in;)

    method:getScale(public; std::vector<double>;)
    { return make_vector(2, 1.0, 1.0); }

    method:getTranslation(public; std::vector<double>;)
    { return make_vector(2, 0.0, -(2.0+1.5*((float) instance))); }

    mode:waiting_for_orders
    {                                                    // mode:waiting_for_orders
      node:receive[order:ord][report:out=>(me;):(sub_times[0])]
      {                                 // node:receive[order:ord][order:out[4]]
        sub_times.push_back(getTime()+random.nextDouble(ord.data));
      }                                 // node:receive[order:ord][order:out[4]]
    }                                                    // mode:waiting_for_orders
  }                                                        // process:soldier(unit)
}
```

C.4.9. squad.proc

```
{import process {unit, soldier} }
{import message {StartSimulation, SetColor} }
{import std {<string>, <iostream>} }

{
  process:squad(unit)
  {                                                          // process:squad(unit)
    soldier:subordinates[10];
    method:init(protected; void;)
    {                                         // method:init(protected; void;)
      unit::init();                               // Call parent class version
      subs = subordinates;                            // Copy the subordinates
      sub_labels.push_back(" Leader");             // Subordinate label values
      sub_labels.push_back(" Radioman");
      sub_labels.push_back(" Heavy gunner 1");
      sub_labels.push_back(" Heavy gunner 2");
      sub_labels.push_back(" Corpsman");
      sub_labels.push_back(" Demolitionist");
      sub_labels.push_back(" Infantryman 1");
      sub_labels.push_back(" Infantryman 2");
      sub_labels.push_back(" Infantryman 3");
      sub_labels.push_back(" Infantryman 4");
    }                                         // method:init(protected; void;)
  }                                                     // process:squad(unit)
}
```

C.4.10. unit.proc

```
{import process {Node2D} }
{import message {report, order, set_parent, SetColor, AddShape2D,
                AddNode2D, SetLabel, SetAffine2D, StartSimulation} }
{import std {<vector>, <iostream>, <string>} }
{import {<stdlib.h>, <time.h>} }

{
  process:unit
  {
    int:instance;                         // Subordinate instance of this unit
    int:units_done(0);              // Is the entire unit done with it's task?
    process:parent;                                            // Parent unit
    process:subs[];                           // Handles to subordinate units
    std::string:label("");                                 // Label for this unit
    std::string:sub_labels[];                 // Labels for subordinate units
    double:sub_times[];                     // Timestamp for subordinate units
    Node2D:unit_node;                          // Graphics node for this unit

    method:setLabel(protected; void; std::string:l;) { label = l; }
    method:getLabel(protected; std::string;) { return label; }

    method:fossilCollect(protected; void;)
    {                                  // method:fossilCollect(protected; void;)
      char ch;

      if (getTime() >= 0.0)                  // If the timestamp is not negative
      {
        if (waiting_for_orders.isActive())          // Unit waiting for orders?
        {
          std::cout << getTime() << ":" << std::endl;
          std::cout << getLabel() << " issuing orders:" << std::endl;
          for (int i=0; i<subs.size(); ++i)
            std::cout << "\t" << sub_labels[i] << " -- " << sub_times[i]
                      << std::endl;
        }                               // if (waiting_for_orders.isActive())
        else if (working.isActive() && units_done==subs.size())
        {
          std::cout << getTime() << ":" << std::endl;
          std::cout << getLabel() << " reports objective achieved."
                    << std::endl << std::endl;
        }                                     // else if (working.isActive())
      }                                              // if (getTime() >= 0.0)
    }                                  // method:fossilCollect(protected; void;)

    method:getScale(public; std::vector<double>;)
    { return make_vector(2, 0.20, 1.0); }

    method:getTranslation(public; std::vector<double>;)
    { return make_vector(2, 150.0*((double) (instance-1.5)), -1.5); }

    mode:start
```

```
{                                                                     // mode:Default
  node:startSimulation[StartSimulation:in]
                     [SetColor:sc=>(unit_node;)]
  { sc.set(1.0, 0.0, 0.0, 1.0); }

  node:setParent[set_parent:in]              // Upon reciept of set_parent
               [set_parent:out[]=>(subs[@];),         // Send subordinates
                AddShape2D:as=>(unit_node;),        // Add in.rect to node
                AddNode2D:an=>(in.parent_node;),        // Add to parent
                SetAffine2D:sat=>(unit_node;),           // Set transform
                SetLabel:sl=>(unit_node;)]         // Set the unit's label
  {                      // node:setParent[set_parent:in][set_parent:out[]]
    parent=in.getSource();                          // Set the parent handle
    instance = in.instance;               // Set subordinate instance number
    setLabel(in.label);                              // Set this instance label
    sl.set(in.label);                   // Label value to set for the unit node

    as.add(in.rect);                 // Add shape as a subordinate to the node
    an.add(unit_node);               // Add unit_node as subordinate to parent

    sat.setScale(getScale());                    // Set the node scaling factor
    sat.setTranslation(getTranslation());           // Set node translation

    if (getType() != SPT_soldier)                         // If not a soldier
    {
      for (int i=0; i<subs.size(); ++i)        // Loop over output messages
      {
        out.push_back(me);                       // Create a new output message
        out.back().instance = i;                   // Set the instance number
        out.back().rect = in.rect;                      // Pass the info along
        out.back().parent_node = unit_node;         // Pass this along, too
        out.back().label = label+sub_labels[i];         // Set label number
      }                                       // for (int i=0; i<subs.size(); ++i)
    }                                            // if (getType() != SPT_soldier)

    start.setActive(false);                      // Deactivate the start mode
    waiting_for_orders.setActive(true);          // Actvt waiting_for_orders
    an.setTX(getType() != SPT_brigade);                     // To avoid a loop
  }                      // node:setParent[set_parent:in][set_parent:out[]]
}                                                                 // mode:Default

mode:waiting_for_orders
{                                                      // mode:waiting_for_orders
  node:startSimulation[StartSimulation:in][]
  { waiting_for_orders.setActive(false); }

  node:receive[order:ord]
            [order:out[]=>(subs[@];):(sub_times[@]),
             SetColor:sc=>(unit_node;)]
  {                              // node:receive[order:ord][order:out[4]]
    sc.set(1.0, 1.0, 0.0);                           // Set the color to yellow

    if (getType() != SPT_soldier)
    {
      for (int i=0; i<subs.size(); ++i)           // Loop over subordinates
      {
        sub_times.push_back(getTime()+random.nextDouble(ord.data));
        out.push_back(me);                               // Create a new order
        out.back().data=ord.data+random.nextDouble(ord.data)/2.0;
      }                                       // for (int i=0; i<subs.size(); ++i)
    }                                            // if (getType() != SPT_soldier)
    waiting_for_orders.setActive(false);              // No longer waiting
    working.setActive(true);                             // We are now working
  }                              // node:receive[order:ord][order:out[4]]
}                                                      // mode:waiting_for_orders
```

C.5. Brigade2

C.5.1. battalion.proc

```
{import process {unit, company} }

{
  process:battalion(unit)
  {                                                 // process:battalion(unit)
    company:subordinates[4];                           // Subordinate objects

    method:init(protected; void;)
    {                                         // method:init(protected; void;)
      unit::init();                              // Call parent class version
      subs = subordinates;                     // Copy the subordinate handles
      sub_labels.push_back(" Alpha Company");
      sub_labels.push_back(" Bravo Company");
      sub_labels.push_back(" Charlie Company");
      sub_labels.push_back(" Delta Company");
    }                                         // method:init(protected; void;)
  }
}
```

C.5.2. brigade.proc

```
{import process {unit, battalion} }
{import message {order, set_parent, StartSimulation} }
{import std {<string>, <iostream>} }

{
  process:brigade(unit)
  {
    battalion:subordinates[4];

    method:init(protected; void;)
    {                                         // method:init(protected; void;)
      unit::init();                              // Call parent class version
      sub_count = 4;                             // We have four subordinates
      subs = subordinates;                     // Copy the subordinate handles
      sub_labels.push_back(" 1st Battalion");
      sub_labels.push_back(" 2nd Battalion");
      sub_labels.push_back(" 3rd Battalion");
      sub_labels.push_back(" 4th Battalion");
    }                                         // method:init(protected; void;)

    mode:start
    {                                                             // mode:start
      node:start[StartSimulation:strt]         // Initial message from engine
                [order:out=>(me;):(0.0),       // Actually start sim at time 0
                 set_parent:sp=>(me;)]         // Set parent for subordinates
      {                               // node:start[StartSimulation:strt][...]
        sp.instance = 0;                            // Set the instance number
        sp.label = "Brigade";                       // Set this instance label
        out.data = 5.0+random.nextDouble(5.0);          // Generate data value
      }                               // node:start[StartSimulation:strt][...]
    }                                                             // mode:start
  }
}
```

C.5.3. company.proc

```
{import process {unit, platoon} }
{import std {<string>, <iostream>} }

{
  process:company(unit)
  {                                                   // process:company(unit)
    platoon:subordinates[4];
    method:init(protected; void;)
    {                                               // method:init(protected; void;)
      unit::init();                                      // Call parent class version
      subs = subordinates;                           // Copy the subordinate handles
      sub_labels.push_back(" 1st Platoon");               // Subordinate label values
      sub_labels.push_back(" 2nd Platoon");
      sub_labels.push_back(" 3rd Platoon");
      sub_labels.push_back(" 4th Platoon");
    }                                               // method:init(protected; void;)
  }                                                        // process:company(unit)
}
```

C.5.4. order.msg

```
{
  message:order
  {
    double:data(0.0);
  }
}
```

C.5.5. platoon.proc

```
{import process {unit, squad} }
{import std {<string>, <vector>, <iostream>} }
{
  process:platoon(unit)
  {                                                        // process:platoon(unit)
    squad:subordinates[4];

    method:init(protected; void;)
    {                                               // method:init(protected; void;)
      unit::init();                                      // Call parent class version
      subs = subordinates;                           // Copy the subordinate handles
      sub_labels.push_back(" Squad 1");                   // Subordinate label values
      sub_labels.push_back(" Squad 2");
      sub_labels.push_back(" Squad 3");
      sub_labels.push_back(" Squad 4");
    }                                               // method:init(protected; void;)
  }                                                        // process:platoon(unit)
}
```

C.5.6. report.msg

```
{
  message:report
  {
    long:instance(0);
  }
}
```

C.5.7. set_parent.msg

```
{import std {<string>} }

{
  message:set_parent
  {                                                         // message:set_parent
    int:instance;             // Subordinate instance number of the destination
    std::string:label;                                  // Label of the destination
  }                                                         // message:set_parent
}
```

C.5.8. soldier.proc

```
{import process {unit} }
{import message {order, report} }
{import std {<string>, <iostream>} }
```

```
{
  process:soldier(unit)
  {
    method:init(protected; void;)
    {                                         // method:init(protected; void;)
      unit::init();                              // Call parent class version
      sub_count = 1;                           // Soldier has no subordinates
      subs.push_back(me);                     // Copy the subordinate handles
    }                                         // method:init(protected; void;)

    method:fossilCollect(protected; void;)
    {                 // method:statusReport(protected; void; sim:report&:in;)
      if (getTime() >= 0.0)                  // If the timestamp is not negative
      {
        if (waiting_for_orders.isActive())        // If unit waiting for orders
        {
          std::cout << getTime() << ":" << std::endl;
          std::cout << getLabel() << " following orders: "
                    << sub_times[0] << std::endl << std::endl;
        }                               // if (waiting_for_orders.isActive())
        else                                          // If it's anything else
          unit::fossilCollect();           // Let parent class take care of it
      }                                               // if (getTime() >= 0.0)
    }                 // method:statusReport(protected; void; sim:report&:in;)

    mode:waiting_for_orders
    {                                                 // mode:waiting_for_orders
      node:receive[order:ord]
              [report:out=>(me;):(sub_times[0])]
      {                              // node:receive[order:ord][order:out[4]]
        sub_times.push_back(getTime()+random.nextDouble(ord.data));
        subs_done.push_back(false);           // The subordinates are not done
      }                              // node:receive[order:ord][order:out[4]]
    }                                                 // mode:waiting_for_orders
  }                                                    // process:soldier(unit)
}
```

C.5.9. squad.proc

```
{import process {unit, soldier} }
{import std {<string>, <iostream>} }

{
  process:squad(unit)
  {                                                      // process:squad(unit)
    soldier:subordinates[10];

    method:init(protected; void;)
    {                                         // method:init(protected; void;)
      unit::init();                              // Call parent class version
      subs = subordinates;                          // Copy the subordinates
      sub_labels.push_back(" Leader");           // Subordinate label values
      sub_labels.push_back(" Radioman");
      sub_labels.push_back(" Heavy gunner 1");
      sub_labels.push_back(" Heavy gunner 2");
      sub_labels.push_back(" Corpsman");
      sub_labels.push_back(" Demolitionist");
      sub_labels.push_back(" Infantryman 1");
      sub_labels.push_back(" Infantryman 2");
      sub_labels.push_back(" Infantryman 3");
      sub_labels.push_back(" Infantryman 4");
    }                                         // method:init(protected; void;)
  }                                                      // process:squad(unit)
}
```

C.5.10. unit.proc

```
{import message {report, order, set_parent, StartSimulation} }
{import std {<vector>, <iostream>, <string>} }
{import {<stdlib.h>, <time.h>} }
```

```
{
  process:unit
  {
    int:instance;                          // Subordinate instance of this unit
    int:sub_count;                                        // # of subordniates
    bool:unit_done(false);           // Is the entire unit done with it's task?
    bool:subs_done[];               // Subordinates reporting that they're done
    process:parent;                                             // Parent unit
    process:subs[];                            // Handles to subordinate units
    std::string:label("");                                // Label for this unit
    std::string:sub_labels[];                  // Labels for subordinate units
    double:sub_times[];                     // Timestamp for subordinate units

    method:setLabel(protected; void; std::string:l;) { label = l; }
    method:getLabel(protected; std::string;) { return label; }

    method:fossilCollect(protected; void;)
    {                                 // method:fossilCollect(protected; void;)
      if (getTime() >= 0.0)                // If the timestamp is not negative
      {
        if (waiting_for_orders.isActive())        // If unit waiting for orders
        {
          std::cout << getTime() << ":" << std::endl;
          std::cout << getLabel() << " issuing orders:" << std::endl;
          for (int i=0; i<sub_count; ++i)
            std::cout << "\t" << sub_labels[i] << " -- " << sub_times[i]
                      << std::endl;
        }                              // if (waiting_for_orders.isActive())
        else if (working.isActive() && unit_done)          // If unit is done
        {
          std::cout << getTime() << ":" << std::endl;
          std::cout << getLabel() << " reports objective achieved."
                    << std::endl << std::endl;
        }                                   // else if (working.isActive())
      }                                            // if (getTime() >= 0.0)
    }                                 // method:fossilCollect(protected; void;)

    mode:start
    {                                                         // mode:Default
      node:setParent[set_parent:in]
                    [set_parent:out[]=>(subs[@];)]
      {                 // node:setParent[set_parent:in][set_parent:out[]]
        parent=in.getSource();                        // Set the parent handle
        instance = in.instance;           // Set the subordinate instance number
        setLabel(in.label);                           // Set this instance label

        if (getType() != SPT_soldier)                         // If not a soldier
        {
          for (int i=0; i<sub_count; ++i)          // Loop over output messages
          {
            out.push_back(me);                     // Create a new output message
            out.back().instance = i;                   // Set the instance number
            out.back().label = label+sub_labels[i];           // Set label number
          }                                   // for (int i=0; i<sub_count; ++i)
        }                                       // if (getType() != SPT_soldier)

        start.setActive(false);                     // Deactivate the start mode
        waiting_for_orders.setActive(true);         // Actvt waiting_for_orders

      }                 // node:setParent[set_parent:in][set_parent:out[]]
    }                                                         // mode:Default

    mode:waiting_for_orders
    {                                              // mode:waiting_for_orders
      node:startSimulation[StartSimulation:in][]
      { waiting_for_orders.setActive(false); }

      node:receive[order:ord]
                 [order:out[]=>(subs[@];):(sub_times[@])]
      {                               // node:receive[order:ord][order:out[4]]
        if (getType() != SPT_soldier)
        {
          for (int i=0; i<sub_count; ++i)         // Loop over the subordinates
          {
            sub_times.push_back(getTime()+random.nextDouble(ord.data));
            subs_done.push_back(false);            // Subordinates are not done
            out.push_back(me);                             // Create a new order
            out.back().data=ord.data+random.nextDouble(ord.data)/2.0;
          }                                   // for (int i=0; i<sub_count; ++i)
        }                                     // for (int i=0; i<sub_count; ++i)
        waiting_for_orders.setActive(false);          // Not waiting for orders
        working.setActive(true);                          // We are now working
      }                              // node:receive[order:ord][order:out[4]]
    }                                                // mode:waiting_for_orders

    mode:working
    {                                                           // mode:working
      node:startSimulation[StartSimulation:in][]
      { working.setActive(false); }

      node:status[report:in][report:out=>(parent;)]
      {                                   // node:status[report:in][report:out]
        unit_done = true;                     // Initialize the unit_done flag

        out.instance = instance;                         // Specify the instance
        subs_done[in.instance] = true;              // This subordinate is done

        for (int i=0; i<subs_done.size(); ++i)        // Loop over subordinates
          unit_done = unit_done && subs_done[i];         // Accumulate the value

        out.setTX(unit_done && parent!=me);           // Report to the parent?
      }                                   // node:status[report:in][report:out]
    }                                                           // mode:working
  }                                                            // process:unit
}
```

C.6. Hierarchy

C.6.1. generic.msg

```
{
  message:generic
  {
    ulong:index(0);
    method:set(public; void; ulong:i;) { index = i; }
    method:get(public; ulong;) { return index; }
  }
}
```

C.6.2. hierarchy.proc

```
{import process {View2D, Polygon2D, Node2D, Vertex2D} }
{import message {SetVertex2D, AddNode2D, AddShape2D, StartSimulation,
                 AddVertex2D, SetScale2D, SetTranslation2D, SetColor,
                 SetLabel, SetMode, generic, SetRefresh, SetSize} }
{import {<GL/glut.h>} }

{
  process:hierarchy
  {                                                        // process:hierarchy
    View2D:view;
    Node2D:nodes[511];
    Polygon2D:rect;
    Vertex2D:vert[4];

    bool:completed[];

    method:init(public; void;) { completed.resize(nodes.size(),false); }

    mode:Default
```

```
{                                                          // mode:Default
 node:start[StartSimulation:in]
           [SetVertex2D:out_sv[4]=>(vert[@];),
            AddNode2D:out_an[]=>( (@==0 ? view : nodes[@-1]); ),
            AddShape2D:out_as=>(nodes;),
            SetScale2D:out_ss=>(nodes;),
            SetTranslation2D:out_st[2],
            AddVertex2D:out_av=>(rect;),
            SetMode:out_sm=>(rect;),
            SetRefresh:out_sr=>(view;),
            SetSize:out_size=>(view;),
            generic:out=>(me;):(0.0)]
 {
   out_sv[0].set(-100.0, -0.5);
   out_sv[1].set( 100.0, -0.5);
   out_sv[2].set( 100.0,  0.5);
   out_sv[3].set(-100.0,  0.5);

   out_an.push_back(me);
   out_an.back().add(nodes[0]);

   for (int i=0; (i+1)*2<nodes.size(); ++i)
   {
     out_an.push_back(me);
     out_an.back().add(nodes[i*2+1]);
     out_an.back().add(nodes[i*2+2]);
   }

   out_as.add(rect);

   out_ss.set(0.5, 1.0);

   out_st[0].set( -150, -1.5);  out_st[1].set( 150, -1.5);

   for (int i=1; i<nodes.size(); ++i)
     out_st[i%2].addDest(nodes[i]);

   out_av.add(vert[0]); out_av.add(vert[1]);
   out_av.add(vert[2]); out_av.add(vert[3]);

   out_sr.set(0.5);

   out_size.set(1000, 600);

   out_sm.set(GL_QUADS);
 }

 node:run[generic:in]
        [generic:out=>(me;):(getTime()+1.0),
         SetColor:out_sc=>(nodes[in.get()];)]
 {                            // node:run[generic:in][generic:out=>(me;)]

   ulong index = in.get();
   ulong left = index*2+1;
   ulong right = index*2+2;
   ulong parent = (index-1)/2;

   if (right > nodes.size() ||
       (completed[left] && completed[right]))
   {
     std::cout << "parent in.get() = " << in.get() << std::endl;

     completed[index] = true;

     out.set(parent);
     out.setTX(index!=0);

     out_sc.set(0.0, 1.0, 0.0, 1.0);
   }
   else if (!completed[left])
   {
          std::cout << "  left in.get() = " << in.get() << std::endl;

          out.set(left);
          out_sc.set(0.0, 0.0, 1.0, 1.0);
        }
        else if (!completed[right])
        {
          std::cout << " right in.get() = " << in.get() << std::endl;

          out.set(right);
          out_sc.set(0.0, 1.0, 1.0, 1.0);
        }
      }                            // node:run[generic:in][generic:out=>(me;)]
    }                                                             // mode:Default
  }                                                          // process:hierarchy
}
```

C.7. Ping

C.7.1. Generic.msg

```
{ message:Generic; }
```

C.7.2. Ping.proc

```
{import std {<iostream>} }

{import message {Generic, StartSimulation} }
{import process {Pong} }

{
  process:Ping
  {                                                           // process:Ping
    int:count(-1);                   // Count of the number of messages received
    Pong:pong;                                  // Something to send a message to

    method:init(public; void;) { std::cout.precision(16); }

    method:fossilCollect(public; void;)
    {                                     // method:fossilCollect(public; void;)
       if (getTime() >= 0)
        std::cout << "Ping (" << count << ") at time " << getTime()
                  << std::endl;
    }                                     // method:fossilCollect(public; void;)

    mode:start
    {                                                           // mode:start
      node:start[StartSimulation:strt]
                [Generic:out=>(me;):(0.0)]
      {               // node:start[StartSimulation:nm][Generic:out=>(pong;)]
        start.setActive(false);                    // Deactivate the start mode
      }               // node:start[StartSimulation:nm][Generic:out=>(pong;)]
    }                                                           // mode:start

    mode:run
    {                                                             // mode:run
      node:ponger[Generic:in][Generic:out=>(pong;)]
      { out.setTX( ++count < 20 ); }
    }                                                             // mode:run
  }                                                           // process:Ping
}
```

C.7.3. Pong.proc

```
{import message {Generic} }
{import std {<iostream>} }

{
  process:Pong
  {                                                           // process:Pong
    int:count(-1);                  // How many times have we received a message

    method:fossilCollect(public; void;)
    {                                       // method:fossilCollect(public; void;)
      if (getTime() >= 0)
        std::cout << "Pong (" << count << ") at time " << getTime()
                  << std::endl;
    }                                       // method:fossilCollect(public; void;)

    mode:Default
    {                                                         // mode:Default
      node:pinger[Generic:in][Generic:out=>(in.getSource();)]
      { out.setTX(++count<20); }
    }                                                         // mode:Default
  }                                                           // process:Pong
}
```

C.8. Relay1

C.8.1. generic.msg

```
{ message:generic; }
```

C.8.2. reflector.proc

```
{import message {generic} }

{
  process:reflector
  {                                                      // process:reflector
    int:count(-1);                  // Count of the number of messages received

    method:init(public; void;) { std::cout.precision(15); }

    method:fossilCollect(public; void;)
    {                                       // method:fossilCollect(public; void;)
      if (count % 10000 == 0)                          // Every 10000 messages
        std::cout << count << " on <" << me.getNode() << ", "
                  << me.getIndex() << "> at time " << getTime()
                  << std::endl;             // Display the message to std::cout
    }                                       // method:fossilCollect(public; void;)

    mode:Default
    {                                                         // mode:Default
      node:reflect[generic:in]          // Upon reciept of a generic input msg
                  [generic:out=>(in.getSource();)]          // Reflect one back
      { count++; }
    }                                                         // mode:Default
  }                                                      // process:reflector
}
```

C.8.3. relay.proc

```
{import message {generic, StartSimulation} }
{import process {reflector} }

{
  process:relay(reflector)
  {
    reflector:r:1;                          // Something to send a message to

    mode:Default
    {                                                         // mode:Default
      node:start[StartSimulation:strt]
                [generic:out=>(r;):(0.0)] { }
    }                                                         // mode:Default
  }
}
```

C.9. Relay2

C.9.1. generic.msg

```
{ message:generic; }
```

C.9.2. reflector.proc

```
{import message {generic, set_partner} }

{
  process:reflector
  {                                                    // process:reflector
    int:count(-1);                  // Count of the number of messages received
    double:t1;                                 // Timestamp of output message 1
    double:t2;                                 // Timestamp of output message 2
    process:partner;                                          // Remote partner

    method:fossilCollect(public; void;)
    {                                    // method:fossilCollect(public; void;)
      if (count >= 0)                     // Is this valid to do at this point?
      {
        std::cout << count << " on " << me << " at time " << getTime()
                  << std::endl;
      }                                                   // if (count >= 0)
    }                                    // method:fossilCollect(public; void;)

    mode:Default
    {                                                          // mode:Default
      node:setPartner[set_partner:in][] { partner = in.getSource(); }

      node:reflect[generic:in]          // Upon reciept of a generic input msg
                  [generic:out1=>(partner;):(t1),          // Reflect one back
                   generic:out2=>(me;):(t2)]        // Send something back here
      {                             // node:reflect[generic][generic, logmsg]
        t1 = getTime()+random.nextDouble(10.0);         // Get output timestamp
        t2 = getTime()+random.nextDouble(10.0);         // Get output timestamp
        count++;                                          // Log the reflection
      }                             // node:reflect[generic][generic, logmsg]
    }                                                          // mode:Default
  }                                                       // process:reflector
}
```

C.9.3. relay.proc

```
{import message {generic, set_partner, StartSimulation} }
{import process {reflector} }

{
  process:relay(reflector)
  {
    reflector:r:1;                        // Something to send a message to

    mode:Default
    {                                                          // mode:Default
      node:start[StartSimulation:strt]
                [generic:out=>(r;):(0.0),
                 set_partner:sp=>(r;):(-0.9)]
        {partner=r;}
    }                                                          // mode:Default
  }
}
```

C.9.4. SetPartner.msg

```
{message:set_partner;}
```

C.10. Relay3

C.10.1. child.proc

```
{import message {generic, setup} }

{
  process:child
  {                                                          // process:child
    long:count(-1);                 // Count of the number of messages received
    ulong:cc;                                            // Number of children
    ulong:ec;                                             // Number of engines
    process:dest;                                       // Destination process
    double:ts;                                   // Outgoing message timestamp

    method:fossilCollect(public; void;)
    {                                 // method:fossilCollect(public; void;)
      if (count % 200 == 0)
        std::cout << count << " on " << me << " at time " << getTime()
                  << std::endl;
    }                                 // method:fossilCollect(public; void;)

    mode:start
    {                                                           // mode:start
      node:relay[setup:in][]
      {                                             // node:relay[setup:in][]
        cc = in.getChildCount();                         // Number of children
        ec = in.getEngineCount();                         // Number of engines
        start.setActive(false);                     // Turn off the start mode
      }                                             // node:relay[setup:in][]
    }                                                           // mode:start

    mode:run
    {                                                             // mode:run
      node:relay[generic:in]
                [generic:out=>(dest;):(ts)]
      {                              // node:relay[generic:in][generic:out]
        ulong di = random.nextInteger(cc);                // Destination index
        dest = process(di%(ec-1)+1, di/ec);       // Set destination processid
        ts = getTime()+random.nextDouble(1.0);                    // Timestamp
        count++;                                         // Log the reflection
      }                              // node:relay[generic:in][generic:out]
    }                                                             // mode:run
  }                                                          // process:child
}
```

C.10.2. generic.msg

```
{ message:generic; }
```

C.10.3. relay.proc

```
{import message {StartSimulation, generic, setup} }
{import process {child} }

{
  process:relay
  {
    child:children[1000]:@%100+1;

    mode:Default
    {                                                         // mode:Default
      node:start[StartSimulation:strt]
                [setup:s=>(children;),
                 generic:out=>(children;):(0.0)]
      {                      // node:start[StartSimulation][setup, generic]
        s.set(children.size(), EngineStand::stand.engineCount()-1);
      }                      // node:start[StartSimulation][setup, generic]
    }                                                         // mode:Default
  }
}
```

C.10.4. setup.msg

```
{
  message:setup
  {
    ulong:childCount;
    ulong:engineCount;
    method:set(public; void; ulong:c; ulong:e;)
      {childCount=c; engineCount=e;}
    method:getChildCount(public; ulong;) { return childCount; }
    method:getEngineCount(public; ulong;) { return engineCount; }
  }
}
```

C.11. Relay4

C.11.1. child.proc

```
{import message {generic, setup, subscribe, unsubscribe} }

{
  process:child
  {                                                      // process:child
    long:count(-1);              // Count of the number of messages received
    ulong:ct;                                         // Number of children
    long:di[3];                                      // Destination index
    double:ts[3];                            // Outgoing message timestamp
    process:subscriptions[];               // Subscription process handles

    method:fossilCollect(public; void;)
    {                                 // method:fossilCollect(public; void;)
      if (count%100 == 0)                                // For the run phase
        std::cout << count << " on " << me << " at time " << getTime()
                  << std::endl;
    }                                 // method:fossilCollect(public; void;)

    mode:start
    {                                                         // mode:start
      node:relay[setup:in]
                [subscribe:sub=>(subscriptions[di[1]];):(ts[1])]
      {                               // node:relay[setup:in][Subscribe:sub]
        subscriptions = in.get();          // Get the subscription handles
        ct = ((ulong) subscriptions.size());    // Number of subscriptions
        di[1]=random.nextInteger(ct);       // Determine subscription to join
        ts[1] = -0.9;                   // Timestamp for the subscription event
        start.setActive(false);                   // Turn off the start mode
      }                               // node:relay[setup:in][Subscribe:sub]
    }                                                         // mode:start

    mode:run
    {                                                           // mode:run
      node:relay[generic:in]                    // Generic inbound message
                [generic:out=>(subscriptions[di[0]];):(ts[0]),
                 subscribe:sub=>(subscriptions[di[1]];):(ts[1]),
                 unsubscribe:unsub=>(subscriptions[di[2]];):(ts[2])]
      {                                       // node:relay[generic:in][ … ]
        for (long i=0; i<3; ++i)                    // Loop over the messages
        {
          di[i] = random.nextInteger(ct);               // Get the destination
          ts[i]=getTime()+random.nextDouble(1.0);          // Get timestamp
        }                                             // for (i=0; i<3; ++i)
        count++;                                       // Log the reflection
      }                                       // node:relay[generic:in][ … ]
    }                                                           // mode:run
  }                                                        // process:child
}
```

C.11.2. generic.msg

```
{ message:generic; }
```

C.11.3. relay.proc

```
{import message {StartSimulation, generic, setup} }
{import process {child, subscription} }

{
  process:relay
  {                                                   // process:relay
    subscription:sub[2];            // Subscription instances for the program
    child:children[4]:@;                              // Child processes

    mode:Default
    {                                                  // mode:Default
      node:start[StartSimulation:strt]
                 [setup:s=>(children;),
                  generic:out=>(sub;):(0.0)]
      {                       // node:start[StartSimulation][setup, generic]
        s.set(sub);                                // Set the subscription
      }                       // node:start[StartSimulation][setup, generic]
    }                                                  // mode:Default
  }                                                   // process:relay
}
```

C.11.4. setup.msg

```
{import std {<vector>} }

{
  message:setup
  {                                                   // message:setup
    process:sub[];                              // Subscription reference
    method:set(public; void; process:s[];) { sub = s; }
    method:get(public; std::vector<process>;) { return sub; }
  }                                                   // message:setup
}
```

C.11.5. subscribe.msg

```
{message:subscribe;}
```

C.11.6. subscription.proc

```
{import message {generic, subscribe, unsubscribe} }
{import std {<set>} }

{
  process:subscription
  {                                             // process:subscription
    std::set<process>:subscribers;           // All of the subscription

    mode:Default
    {                                                  // mode:Default
      node:subscribe[subscribe:in][]
        { subscribers.insert(in.getSource()); }

      node:unsubscribe[unsubscribe:in][]
        { subscribers.erase(in.getSource()); }

      node:forward[generic:in][generic:out[]]          // Forward generic msg
      {                           // node:forward[generic:in][generic:out[]]
        out.push_back(in);                       // Copy the input message
        out.back().clearDest();               // Clear the destination list
        std::set<process>::iterator i;              // subscribers iterator
        for (i=subscribers.begin(); i!=subscribers.end(); ++i)
          out.back().addDest(*i);            // Change the destination list
      }                           // node:forward[generic:in][generic:out[]]
    }                                                    // mode:Default
  }                                              // process:subscription
}
```

C.11.7. unsubscribe.msg

```
{message:unsubscribe;}
```

C.12. Relay5

C.12.1. base.proc

```
{import message {generic} }
{import std {<string>} }

{
  process:base
  {                                                    // process:base
    long:count(-1);                                         // Counter
    std::string:label;                    // Label for this base instance

    method:init(public; void;) { std::cout.precision(15); }

    method:fossilCollect(public; void;)
    {                               // method:fossilCollect(public; void;)
      if (count%1000 == 0)
        std::cout << label << ": (" << count << ") @ " << getTime()
                  << std::endl;
    }                               // method:fossilCollect(public; void;)

    mode:Default
    {                                                    // mode:Default
      node:routine[generic:in][] { ++count; }       // Increment counter
    }                                                    // mode:Default
  }                                                      // process:base
}
```

C.12.2. generic.msg

```
{ message:generic; }
```

C.12.3. relay.proc

```
{import message {generic} }
{import process {base, sink} }

{
  process:relay(base)
  {
    sink:s:me.getNode()+1;                // Sink for the message stream

    method:init(public; void;) { base::init(); label = " relay"; }

    mode:Default
    {                                                    // mode:Default
      node:run[generic:in][generic:out=>(s;)] {}
    }                                                    // mode:Default
  }
}
```

C.12.4. sink.proc

```
{import process {base} }

{
  process:sink(base)
  {                                               // process:sink(base)
    method:init(public; void;) { base::init(); label = "  sink"; }
  }                                               // process:sink(base)
}
```

C.12.5. source.proc

```
{import message {StartSimulation, generic} }
{import process {relay, base} }

{
  process:source(base)
  {                                                    // process:source(base)
    relay:r:me.getNode()+1;                                    // Relay process

    method:init(public; void;) { base::init(); label = "source"; }

    mode:Default
    {                                                           // mode:Default
      node:start[StartSimulation:s][generic:out=>(me; r;):(0.0)] {}
      node:run[generic:in][generic:out=>(me; r;)] {}
    }                                                           // mode:Default
  }                                                    // process:source(base)
}
```

C.13. Relay6

C.13.1. base.proc

```
{import message {generic} }
{import std {<string>} }

{
  process:base
  {                                                             // process:base
    long:count(-1);                                                 // Counter
    std::string:label;                            // Label for this base instance

    method:fossilCollect(public; void;)
    {                                   // method:fossilCollect(public; void;)
      if (getTime() >= 0)
      {
        ulong p = std::cout.precision();
        std::cout.precision(12);
        std::cout << label << ": (" << count << ") @ " << getTime()
                  << std::endl;
        std::cout.precision(p);
      }
    }                                   // method:fossilCollect(public; void;)

    method:round(public; double; double:t;)
      { return floor(t*10.0+0.5)/10.0; }

    mode:Default
    {                                                           // mode:Default
      node:routine[generic:in][] { ++count;}               // Increment counter
    }                                                           // mode:Default
  }                                                             // process:base
}
```

C.13.2. generic.msg

```
{ message:generic; }
```

C.13.3. relay.proc

```
{import message {generic, StartSimulation} }
{import process {base, sink} }

{
  process:relay(base)
  {
    sink:s:1;                                      // Sink for the message stream
    ulong:ct(0);                                                      // Counter

    method:init(public; void;) { base::init(); label = " relay"; }

    mode:Default
    {                                                         // mode:Default
      node:start[StartSimulation:in][generic:out=>(me;s;):(0.0)] { }

      node:run[generic:in]
              [generic:out=>(me;):(round(getTime()+0.1))]
      {                          // node:run[generic:in][generic:out=>(me;)]
        if (++ct%10==0) out.addDest(s);             // Do we send it to s, too
      }                          // node:run[generic:in][generic:out=>(me;)]
    }                                                         // mode:Default
  }
}
```

C.13.4. sink.proc

```
{import message {generic} }
{import process {base} }
{import {<math.h>} }

{
  process:sink(base)
  {                                                  // process:sink(base)
    double:lt(-1.0);
    method:init(public; void;) { base::init(); label="  sink"; }

    mode:Default
    {                                                        // mode:Default
      node:run[generic:in][generic:out=>(me;):(round(getTime()+1.0))]
      { out.setTX(fabs(getTime()-lt)>0.1); lt=getTime(); }
    }                                                        // mode:Default
  }                                                  // process:sink(base)
}
```

C.14. Ring1

C.14.1. Generic.msg

```
{import message {ReportValue} }
{message:Generic(ReportValue);}
```

C.14.2. ReportIndex.msg

```
{import message {ReportValue} }
{ message:ReportIndex(ReportValue); }
```

C.14.3. ReportSize.msg

```
{import message {ReportValue} }
{message:ReportSize(ReportValue);}
```

C.14.4. ReportValue.msg

```
{
  message:ReportValue
  {                                                // message:ReportValue
    ulong:value( ((ulong) -1) );              // Size of the subscription
    method:set(public; void; ulong:v;) { value = v; }          // Set the value
    method:get(public; ulong;) { return value; }           // Return the value
  }                                                // message:ReportValue
}
```

C.14.5. Ring.proc

```
{import process {RingMember, Subscription} }
{import message {Generic, StartSimulation, Setup, ReportSize} }

{
  process:Ring
  {                                                           // process:Ring
    RingMember:ring[10];                                 // Ring of elements
    Subscription:sub;                                   // Subscription list

    mode:Default
    {                                                         // mode:Default
      node:start[StartSimulation:strt]                  // Start the simulation
                [Setup:s=>(ring;),               // Broadcast a setup message
                 ReportSize:r=>(sub;):(-0.5),       // Report size to members
                 Generic:out=>(ring[0];):(0.0)]               // Start the ring
      {                              // node:start[StartSimulation:strt][ … ]
        s.set(sub);                            // Set the subscription parameter
      }                              // node:start[StartSimulation:strt][ … ]
    }                                                         // mode:Default
  }                                                           // process:Ring
}
```

C.14.6. RingMember.proc

```
{import message {Generic, Setup, ReportSize, ReportIndex, Subscribe} }

{
  process:RingMember
  {                                                     // process:RingMember
    process:sub;                      // Handle to the subscription process
    long:count(-1);                                       // A simple counter
    long:next(0);                                   // Index of next instance
    long:index(0);                                  // Index of this instance

    method:init(protected; void;) { std::cout.precision(16); }

    method:fossilCollect(public; void;)
    {                                  // method:fossilCollect(public; void;)
      if (getTime() >= 0)
        std::cout << "RingMember[" << index<< "]: (" << count
                  << ") at time " << getTime() << std::endl;
    }                                  // method:fossilCollect(public; void;)

    mode:Default
    {                                                         // mode:Default
      node:setup[Setup:in][Subscribe:out=>(sub;)] { sub=in.get(); }
      node:setIndex[ReportIndex:in][] { index=in.get(); }
      node:setNext[ReportSize:in][] { next = (index+1) % in.get(); }

      node:run[Generic:in][Generic:out => (sub;) ]
      {                                     // node:run[Generic:in][Generic:out]
        out.set(next);            // Set the index of the eventual destination
        out.setTX(count++<10);                    // Do we continue transmitting?
      }                                     // node:run[Generic:in][Generic:out]
    }                                                         // mode:Default
  }                                                     // process:RingMember
}
```

C.14.7. Setup.msg

```
{
  message:Setup
  {                                                          // message:Setup
    process:sub;                             // Subscription reference to use
    method:set(public; void; process:p;) { sub=p; }             // Set the sub
    method:get(public; process;) { return sub; }              // Return the sub
  }                                                          // message:Setup
}
```

C.14.8. Subscribe.msg

```
{message:Subscribe;}
```

C.14.9. Subscription.proc

```
{import message {Subscribe, ReportSize, ReportIndex, Generic} }

{
  process:Subscription
  {                                                      // process:Subscription
    process:subscribers[];                               // List of subscribers

    mode:Default
    {                                                            // mode:Default
      node:subscribe[Subscribe:in][ReportIndex:out=>(in.getSource();)]
      {               // node:subscribe[Subscribe:in][ReportIndex:out]
        out.set(subscribers.size());                // Index value to report back
        subscribers.push_back(in.getSource());             // Add the subscriber
      }               // node:subscribe[Subscribe:in][ReportIndex:out]

      node:reportSize[ReportSize:in][ReportSize:out=>(subscribers;)]
      {               // node:reportSize[ReportSize:in][ReportSize:out]
        out.set(subscribers.size());               // Set size of output message
      }               // node:reportSize[ReportSize:in][ReportSize:out]

      node:forward[Generic:in][Generic:out]
      {                                 // node:forward[Generic:in][Generic:out]
        if (in.get() < subscribers.size())                    // If a valid value
          out.addDest(subscribers[in.get()]);             // Forward only to dest
        else                                           // If the value is not valid
          out.addDest(subscribers);                  // Broadcast to all subscribers
        out.set(in.get());                               // Report the proper index
      }                                 // node:forward[Generic:in][Generic:out]
    }                                                            // mode:Default
  }                                                      // process:Subscription
}
```

C.15. Ring2

C.15.1. Generic.msg

```
{import message {ReportValue} }
{message:Generic(ReportValue);}
```

C.15.2. ReportIndex.msg

```
{import message {ReportValue} }
{ message:ReportIndex(ReportValue); }
```

C.15.3. ReportSize.msg

```
{import message {ReportValue} }
{message:ReportSize(ReportValue);}
```

C.15.4. ReportValue.msg

```
{
  message:ReportValue
  {                                                      // message:ReportValue
    ulong:value( ((ulong) -1) );                     // Size of the subscription
    method:set(public; void; ulong:v;) { value = v; }            // Set the value
    method:get(public; ulong;) { return value; }              // Return the value
  }                                                      // message:ReportValue
}
```

C.15.5. Ring.proc

```
{import process {RingMember, Subscription} }
```

```
{import message {Generic, StartSimulation, Setup, ReportSize} }

{
  process:Ring
  {                                                            // process:Ring
    RingMember:ring[10];                                    // Ring of elements
    Subscription:sub;                                      // Subscription list

    mode:Default
    {                                                          // mode:Default
      node:start[StartSimulation:strt]                  // Start the simulation
                [Setup:s=>(ring;),               // Broadcast a setup message
                 ReportSize:r=>(sub;):(-0.5),         // Report size to members
                 Generic:out=>(sub;):(0.0)]                   // Start the ring
      {                             // node:start[StartSimulation:strt][ … ]
        s.set(sub);                        // Set the subscription parameter
      }                             // node:start[StartSimulation:strt][ … ]
    }                                                          // mode:Default
  }                                                            // process:Ring
}
```

C.15.6. RingMember.proc

```
{import message {Generic, Setup, ReportSize, ReportIndex, Subscribe} }

{
  process:RingMember
  {                                                      // process:RingMember
    process:sub;                           // Handle to the subscription process
    long:count(-1);                                          // A simple counter
    long:next(0);                                        // Index of next instance
    long:index(0);                                       // Index of this instance

    method:init(public; void;) { std::cout.precision(16); }

    method:fossilCollect(public; void;)
    {                                     // method:fossilCollect(public; void;)
      if (getTime() >= 0)
        std::cout << "RingMember[" << index<< "]: (" << count
                  << ") at time " << getTime() << std::endl;
    }                                     // method:fossilCollect(public; void;)

    mode:Default
    {                                                            // mode:Default
      node:setup[Setup:in][Subscribe:out=>(sub;)] { sub=in.get(); }
      node:setIndex[ReportIndex:in][] { index=in.get(); }
      node:setNext[ReportSize:in][] { next = (index+1) % in.get(); }

      node:run[Generic:in][Generic:out => (sub;) ]
      {                                    // node:run[Generic:in][Generic:out]
        out.set(next);            // Set the index of the eventual destination
        out.setTX(count++<10);                      // Do we continue transmitting?
      }                                    // node:run[Generic:in][Generic:out]
    }                                                            // mode:Default
  }                                                      // process:RingMember
}
```

C.15.7. Setup.msg

```
{
  message:Setup
  {                                                          // message:Setup
    process:sub;                             // Subscription reference to use
    method:set(public; void; process:p;) { sub=p; }            // Set the sub
    method:get(public; process;) { return sub; }            // Return the sub
  }                                                          // message:Setup
}
```

C.15.8. Subscribe.msg

```
{message:Subscribe;}
```

C.15.9. Subscription.proc

```
{import message {Subscribe, ReportSize, ReportIndex, Generic} }

{
  process:Subscription
  {                                                     // process:Subscription
    process:subscribers[];                              // List of subscribers

    mode:Default
    {                                                           // mode:Default
      node:subscribe[Subscribe:in][ReportIndex:out=>(in.getSource();)]
      {                          // node:subscribe[Subscribe:in][ReportIndex:out]
        out.set(subscribers.size());                 // Index value to report back
        subscribers.push_back(in.getSource());               // Add the subscriber
      }                          // node:subscribe[Subscribe:in][ReportIndex:out]

      node:reportSize[ReportSize:in][ReportSize:out=>(subscribers;)]
      {                         // node:reportSize[ReportSize:in][ReportSize:out]
        out.set(subscribers.size());                 // Set size of output message
      }                         // node:reportSize[ReportSize:in][ReportSize:out]

      node:forward[Generic:in][Generic:out]
      {                                  // node:forward[Generic:in][Generic:out]
        if (in.get() < subscribers.size())                    // If a valid value
          out.addDest(subscribers[in.get()]);             // Forward only to dest
        else                                          // If the value is not valid
          out.addDest(subscribers);                 // Broadcast to all subscribers
        out.set(in.get());                              // Report the proper index
      }                                  // node:forward[Generic:in][Generic:out]
    }                                                           // mode:Default
  }                                                     // process:Subscription
}
```

C.16. Simple1

C.16.1. Generic.msg

```
{message:Generic;}
```

C.16.2. Simple.proc

```
{import std {<iostream>} }

{import message {Generic, StartSimulation} }

{
  process:Simple
  {                                                           // process:Simple
    int:count(-1);                                               // Hit counter

    method:init(public; void;) { std::cout.precision(15); }

    method:fossilCollect(public; void;)
    {                                    // method:fossilCollect(public; void;)
      if (getTime() >= 0)                 // If this is a good time to proceed
      {
        std::cout << me << ": ";
        if (getTime() == 0) std::cout << "starting";
        else std::cout << count << " @ time " << getTime();
        std::cout << std::endl;
      }                                                 // if (getTime() >= 0)
    }                                    // method:fossilCollect(public; void;)

    mode:start
    {                                                           // mode:start
      node:proc[StartSimulation:strt][Generic:om=>(me;):(0.0)]
      {                   // node:proc[StartSimulation:strt][Generic:om=>(me;)]
        start.setActive(false);                  // Turn the start mode off
      }                   // node:proc[StartSimulation:strt][Generic:om=>(me;)]
    }                                                           // mode:start

    mode:run
    {                                                             // mode:run
      node:proc[Generic:im][Generic:om=>(me;)]
      {                             // node:proc[Generic:im][Generic:om=>(me;)]
        om.setTX(++count < 100);                          // Do we transmit?
      }                             // node:proc[Generic:im][Generic:om=>(me;)]
    }                                                             // mode:run
  }                                                         // process:Simple
}
```

C.17. Simple2

C.17.1. Generic.msg

```
{message:Generic;}
```

C.17.2. Simple.proc

```
{import message {StartSimulation, Generic} }
{import std {<iostream>} }

{
  process:Simple
  {                                                         // process:Simple
    long:count(-1);                                               // Counter

    method:init(public; void;) { std::cout.precision(16); }

    method:fossilCollect(public; void;)
    {                                 // method:fossilCollect(public; void;)
      if (count % 10000 == 0)
        std::cout << "Simple (" << count << ") at time " << getTime()
                  << std::endl;
    }                                 // method:fossilCollect(public; void;)

    mode:start
    {                                                           // mode:start
      node:proc[StartSimulation:in][Generic:out=>(me;):(0.0)]
      { start.setActive(false); }
    }                                                           // mode:start

    mode:run
    {                                                             // mode:run
      node:proc[Generic:in][Generic:out=>(me;)]
        { count++; }                                                // Loop
    }                                                             // mode:run
  }                                                         // process:Simple
}
```

C.18. Simple3

C.18.1. Child.proc

```
{import message {SetParent, Generic} }

{
  process:Child
  {                                                          // process:Child
    process:parent;                           // Handle to the parent process
    long:count(-1);                                               // Counter
    method:init(public; void;) { std::cout.precision(16); }

    method:fossilCollect(public; void;)
    {                                    // method:fossilCollect(public; void;)
      if (run.isActive() && count % 10000==0)           // Every 10000 messages
        std::cout << " Child (" << count << ") at time " << getTime()
                  << std::endl;                           // Display the status
    }                                    // method:fossilCollect(public; void;)

    mode:start
    {                                                            // mode:start
      node:setParent[SetParent:in][]
      {                                       // node:setParent[SetParent:in][]
        parent = in.getSource();                    // Set the parent reference
        start.setActive(false);                           // Turn this mode off
      }                                       // node:setParent[SetParent:in][]
    }                                                            // mode:start

    mode:run
    {                                                          // mode:Default
      node:bounce[Generic:in][Generic:out=>(parent;)] { count++; }
    }                                                          // mode:Default
  }                                                           // process:Child
}
```

C.18.2. Generic.msg

```
{message:Generic;}
```

C.18.3. SetParent.msg

```
{message:SetParent;}
```

C.18.4. Simple.proc

```
{import message {StartSimulation, Generic, SetParent} }
{import process {Child} }
{import std {<iostream>} }

{
  process:Simple
  {                                                          // process:Simple
    long:count(-1);                                                 // Counter
    Child:child;                                              // Child process

    method:init(public; void;) { std::cout.precision(16); }

    method:fossilCollect(public; void;)
    {                                    // method:fossilCollect(public; void;)
      if (run.isActive() && count % 10000==0)           // Every 10000 messages
        std::cout << "Simple (" << count << ") at time " << getTime()
                  << std::endl;                          // Display the status
    }                                    // method:fossilCollect(public; void;)

    mode:start
    {                                                          // mode:Default
      node:start[StartSimulation:strt]
                [Generic:out=>(child;):(0.0), SetParent:sp=>(child;)]
      {           // node:start[StartSimulation:strt][Generic:out,SetParent:sp]
        start.setActive(false);                  // Deactivate the start mode
      }           // node:start[StartSimulation:strt][Generic:out,SetParent:sp]
    }                                                          // mode:Default

    mode:run
    {                                                          // mode:Default
      node:bounce[Generic:in][Generic:out=>(child;)] { count++; }
    }                                                          // mode:Default
  }                                                          // process:Simple
}
```

Appendix D. References

(Atkinson 1989)	K. Atkinson, 1989, *An Introduction to Numerical Analysis, 2nd Edition*, New York, John Wiley & Sons
(Bauer 1992)	H. Bauer, C. Sporrer, 1992, "Distributed Logic Simulation and an Approach to Asynchronous GVT-Calculation", *Proceedings 6th Workshop on Parallel and Distributed Simulation*, vol 24, pp 205-206,
(Bellenot 1990)	S. Bellenot, 1990 "Global virtual time algorithms", *Proceedings of the Multiconference on Distributed Simulation*, 22 (1), January, pp. 122-127.
(Boeing 2001)	The Boeing Company, 2001, online product description of digital design methodologies for the Boeing 777 family of commercial airliners http://www.boeing.com/commercial/777family/cdfacts.html
(Bryant 1977)	R. Bryant, 1977, *Simulation of Packet Communication Architecture Computer Systems*. MIT-LCS-TR-188, Massachusetts Institute of Technology, Cambridge, MA
(Chandy 1979)	K. Chandy, J. Misra, J., 1979, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs", *IEEE Transactions on Software Engineering*, Vol. SE-5, No 5, September, pp. 440-452
(Chandy 1981)	K. Chandy, J. Misra, J., 1981, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", *Communications of the ACM*, 24:4, 198-205
(Concepcion 1990)	A. Concepcion, S. Kelly, 1990, "Computing Global Virtual Time Using the Multi-Level Token Passing Algorithm", *Advances in Parallel and Distributed Simulation*, vol 23, pp 63-70
(Cormen 1990)	T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, MIT-Press, McGraw Hill, Cambridge, MA
(DMSO 2001)	Defense Modeling and Simulation Organization, 2001, online statement of mission and purpose http://www.dmso.mil/index.php?page=133
(DODI 1996)	Department of Defense Instruction (DODI) 5000.61, 29 April 1996, *DoD Modeling and Simulation (M&S) Verification, Validation, and Accreditation (VV&A)*,
(D'Souza 1994)	L. D'Souza, X. Fan, P. A. Wilsey, 1994, "pGVT: An algorithm for accurate GVT estimation", *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, Edinburgh Scotland, pp102-109
(Fabbri 1999)	A. Fabbri, 1999, "GVT and Scheduling in Space Time Memory Based Techniques" *Proceedings 13th Workshop on Parallel and Distributed Simulation*, Atlanta, GA, pp. 54-61
(Fishwick 1995)	P. Fishwick, 1995, *Simulation Model Design and Execution; Building Digital Worlds*, Englewood Cliffs, NJ, Prentice-Hall
(Foley 1996)	J. Foley, et al, 1996 *Computer Graphics: Principles and Practice in C: 2/e*, Reading, MA Addison Wesley Longman
(Fujimoto 1990)	R. Fujimoto, 1990 , "Parallel Discrete Event Simulation", *Communications of the ACM*, Vol. 33 (10), October
(Fujimoto 1993)	R. Fujimoto, 1993, "Parallel Discrete Event Simulation: Will the Field Survive?" *OSRA Journal on Computing*, 5(3):213-230

(Fujimoto 1997) R. Fujimoto, M. Hybinette, 1997, "Computing Global Virtual Time in Shared-Memory Multiprocessors", ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 4, October, pp 425-446

(Gropp 1998) W. Gropp, et al, 1998, *MPI – The Complete Reference, Volume 2, The MPI Extensions*, Cambridge, MA, The MIT Press

(Gropp 1999a) W. Gropp, E. Lusk, A. Skjellum, 1999, *Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd Edition*, Cambridge, MA, The MIT Press

(Gropp 1999b) W. Gropp, E. Lusk, A. Skjellum, 1999, *Using MPI-2: Advanced Features of the Message-Passing Interface*, Cambridge, MA, The MIT Press

(Hockney 1988) R. Hockney, Eastwood James, 1988, *Computer Simulation Using Particles*, Bristol, Adam Hilger

(Hungerford 1974) T. Hungerford, 1974, *Algebra*, New York, Springer Verlag Graduate Texts in Mathematics, pp 7-9

(Isaacson 1966) E. Isaacson, H. Keller, Herbert. 1966 *Analysis of Numerical Methods*, New York, Dover Publications, Inc.

(Jefferson 1982) D. Jefferson, H. Sowizral, 1982, *Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control*, Technical Report N-1906-AF, RAND Corporation, Santa Monica, CA

(Jefferson 1985a) D. Jefferson, 1985, "Virtual Time", *ACM Transactions on Programming Languages and Systems*, 7 (3), July 1985, pp. 3-7

(Jefferson 1985b) D. Jefferson, H. Sowizral, 1985, "Fast concurrent simulation using the Time Warp mechanism", *Proceedings of the Conference on Distributed Simulation*, volume 15(2), San Diego, CA, January, 63-69

(Jefferson 1987) D. Jefferson, et al, 1987, "The Time Warp Operating System." *11th Symposium on Operating Systems Principles* 21, 5, November, 77-93

(Josuttis 1999) N. Josuttis, 1999, *The C++ Standard Library; A Tutorial and Reference*, Reading, MA, Addison Wesley Longman

(Kuhl 1999) F. Kuhl, R. Weatherly, J. Dahmann, 1999, *Creating Computer Simulation Systems: An introduction to the High Level Architecture*, Upper Saddle River, NJ, Prentice Hall PTR

(Lin 1989) Y. Lin, E. Lazowska, 1989, *Determining the global virtual time in distributed simulation*, Technical Report 90-01-02, Department of Computer Science, University of Washington, Seattle, WA

(LLNL 1998) Lawrence Livermore National Laboratory, 2001, online JCATS Executive Summary, http://www.llnl.gov/nai/group/JCATSExecSummary.htm

(Mansuripur 1997) M. Mansuripur, 1997, "The Ronchi Test", *Optics & Photonics News*, July, pp 42-46

(MathWorks 2001) The MathWorks, 2001, online product Literature for Simulink 4, available at http://www.mathworks.com/products/simulink/

(Mattern 1993) F. Mattern,1993, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation", *Journal of Parallel and Distributed Computing*, vol 18, pp 423-434

(Morrison 1991)	F. Morrison, 1991, *The Art of Modeling Dynamic Systems; Forecasting for Chaos, Randomness, & Determinism*, New York, John Wiley & Sons
(Priess 1990)	B. Priess, I. MacIntyre, 1990, *YADDES – Yet Another Distributed Discrete Event Simulator: User Manual*, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada
(Press 1992)	W. Press et al. 1992 *Numerical Recipes in C; The Art of Scientific Computing, 2nd Edition*, Cambridge University Press
(Reiher 1990a)	P. Reiher, D. Jefferson, 1990, "Virtual Time Based Dynamic Load Management in the Time Warp Operating System", *Transactions if the Society for Computer Simulation*, Vol 7(2), June
(Reiher 1990b)	P. Reiher, F. Wieland, P. Hontalas, 1990, "Providing Determinism in the Time Warp Operating System - Costs, Benefits, and Implications", *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, October
(Reiher 1990c)	P. Reiher, 1990, "Parallel Simulation Using the Time Warp Operating System", *Proceedings of the 1990 Winter Simulation Conference*, December
(Reiher 1991a)	P. Reiher, S. Bellenot, D. Jefferson, 1991, "Temporal Decomposition of Simulations Under the Time Warp Operating System", *Proceedings of the 1991 Principles of Distributed Simulation Conference*
(Reiher 1991b)	P. Reiher, S. Bellenot, D. Jefferson, 1991 "Debugging the Time Warp Operating System and Its Applications", *Proceedings of the Symposium in Experiences with Distributed and Multiprocessor Systems II*, March
(Reiher 1992)	P. Reiher, 1992, "Experiences With Optimistic Synchronization for Distributed Operating Systems", *Proceedings of the Third Symposium on Experiences with Distributed and Multiprocessor Systems*, March
(Rosenbloom 1994)	P. Rosenbloom, et al, 1994. "Intelligent automated agents for tactical air simulation: a progress report". *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation.* Orlando, FL.
(STRICOM 1999)	United States Army Simulation, Training and Instrumentation Command, 1999, *Advanced Distributed Simulation Technology II: ModSAF 5.0 Functional Description Document*, ADST-II-CDRL-MODSAF5.0-9800327
(Wonnacott 1996)	P. Wonnacott, 1996, *Run-time Support for Parallel Discrete Event Simulation Languages*, Ph.D. Dissertation, University of Exeter, Faculty of Science
(Wright 1996)	R. Wright, M. Sweet, 1996, *OpenGL Superbible*, Waite Group Press
(Xiao 1995)	Z. Xiao, F. Gomes, B. Unger, 1995, "A Fast Asynchronous GVT Algorithm for Shared Memory Multiprocessor Architectures", *Proceedings of The 1995 Workshop on Parallel and Distributed Simulation*, IEEE Computer Society